pSOSystem Product Family

pSOSystem
System Calls

000-5432-001

Copyright © 1999 Integrated Systems, Inc. All rights reserved. Printed in U.S.A.
Document Title: **pSOSystem System Calls**
Part Number: **000-5432-001**
Revision Date: **January 1999**

Integrated Systems, Inc. • 201 Moffett Park Drive • Sunnyvale, CA 94089-1322

	Corporate	pSOS or pRISM+ Support	$MATRIX_X$ Support
Phone	408-542-1500	1-800-458-7767, 408-542-1925	1-800-958-8885, 408-542-1930
Fax	408-542-1950	408-542-1966	408-542-1951
E-mail	ideas@isi.com	psos_support@isi.com	mx_support@isi.com
Home Page	http://www.isi.com		

LICENSED SOFTWARE - CONFIDENTIAL/PROPRIETARY

This document and the associated software contain information proprietary to Integrated Systems, Inc., or its licensors and may be used only in accordance with the Integrated Systems license agreement under which this package is provided. No part of this document may be copied, reproduced, transmitted, translated, or reduced to any electronic medium or machine-readable form without the prior written consent of Integrated Systems.

Integrated Systems makes no representation with respect to the contents, and assumes no responsibility for any errors that might appear in this document. Integrated Systems specifically disclaims any implied warranties of merchantability or fitness for a particular purpose. This publication and the contents hereof are subject to change without notice.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS252.227-7013 or its equivalent. Unpublished rights reserved under the copyright laws of the United States.

TRADEMARKS

AutoCode, ESp, $MATRIX_X$, pRISM, pRISM+, pSOS, SpOTLIGHT, and Xmath are registered trademarks of Integrated Systems, Inc. BetterState, BetterState Lite, BetterState Pro, DocumentIt, Epilogue, HyperBuild, OpEN, OpTIC, pHILE+, pLUG&SIM, pNA+, pREPC+, pROBE+, pRPC+, pSET, pSOS+, pSOS+m, pSOSim, pSOSystem, pX11+, RealSim, SystemBuild, and ZeroCopy are trademarks of Integrated Systems, Inc.

ARM is a trademark of Advanced RISC Machines Limited. Diab Data and Diab Data in combination with D-AS, D-C++, D-CC, D-F77, and D-LD are trademarks of Diab Data, Inc. ELANIX, Signal Analysis Module, and SAM are trademarks of ELANIX, Inc. SingleStep is a trademark of Software Development Systems, Inc. SNiFF+ is a trademark of TakeFive Software GmbH, Austria, a wholly-owned subsidiary of Integrated Systems, Inc.

All other products mentioned are the trademarks, service marks, or registered trademarks of their respective holders.

Contents

Using This Manual xix

Organization xix

Conventions xx

Font Conventions xx

Symbol Conventions xxi

Format Conventions xxi

Note, Caution, and Warning Conventions xxiii

Revision Bar Convention xxiii

Commonly Used Terms and Acronyms xxiv

Related Publications xxv

Support xxvii

Contacting Integrated Systems Support xxviii

1 pSOSystem System Calls

2 pSOS+ System Calls

as_catch 2-9

as_notify 2-14

as_return 2-16

as_send 2-18
co_register 2-20
co_unregister 2-23
cv_abroadcast 2-25
cv_asignal 2-27
cv_broadcast 2-29
cv_create 2-31
cv_delete 2-33
cv_ident 2-35
cv_info 2-37
cv_signal 2-39
cv_wait 2-41
de_close 2-43
de_cntrl 2-45
de_init 2-47
de_open 2-49
de_read 2-51
de_write 2-53
dnt_add 2-55
dnt_find 2-57
dnt_remove 2-59
errno_addr 2-60
ev_asend 2-62
ev_receive 2-64
ev_send 2-67
ioj_bind 2-69
ioj_bindany 2-71

ioj_getent 2-73
ioj_lock 2-75
ioj_unbind 2-76
ioj_unlock 2-78
k_fatal 2-79
k_terminate 2-81
m_ext2int 2-83
m_int2ext 2-85
mu_create 2-87
mu_delete 2-90
mu_ident 2-92
mu_info 2-94
mu_lock 2-96
mu_setceil 2-98
mu_unlock 2-100
ob_roster 2-102
pt_create 2-105
pt_delete 2-108
pt_getbuf 2-110
pt_ident 2-112
pt_info 2-114
pt_retbuf 2-116
pt_sgetbuf 2-118
q_asend 2-120
q_aurgent 2-122
q_avsend 2-124
q_avurgent 2-126

q_broadcast 2-128
q_create 2-130
q_delete 2-133
q_ident 2-135
q_info 2-137
q_notify 2-139
q_receive 2-141
q_send 2-144
q_urgent 2-146
q_vbroadcast 2-148
q_vcreate 2-150
q_vdelete 2-153
q_vident 2-155
q_vinfo 2-157
q_vnotify 2-159
q_vreceive 2-161
q_vsend 2-163
q_vurgent 2-165
rn_create 2-167
rn_delete 2-170
rn_getseg 2-172
rn_ident 2-174
rn_info 2-176
rn_retseg 2-178
sm_av 2-180
sm_create 2-182
sm_delete 2-185

sm_ident 2-187
sm_info 2-189
sm_notify 2-191
sm_p 2-193
sm_v 2-195
sys_info 2-197
t_addvar 2-201
t_create 2-203
t_delete 2-208
t_delvar 2-211
t_getreg 2-213
t_ident 2-215
t_info 2-217
t_mode 2-221
t_restart 2-225
t_resume 2-227
t_setpri 2-229
t_setreg 2-231
t_start 2-233
t_suspend 2-237
t_tslice 2-239
tm_cancel 2-241
tm_evafter 2-243
tm_evevery 2-245
tm_evwhen 2-247
tm_get 2-250
tm_getticks 2-252

tm_set 2-254
tm_tick 2-256
tm_wkafter 2-258
tm_wkwhen 2-260
tsd_create 2-262
tsd_delete 2-266
tsd_getval 2-268
tsd_ident 2-270
tsd_info 2-272
tsd_setval 2-274

3 pHILE+ System Calls

access_f 3-7
annex_f 3-8
cdmount_vol 3-10
change_dir 3-12
chmod_f 3-14
chown_f 3-16
close_dir 3-17
close_f 3-18
control_vol 3-20
create_f 3-23
fchmod_f 3-25
fchown_f 3-27
fstat_f 3-28
fstat_vfs 3-31
ftruncate_f 3-33

get_fn 3-35
init_vol 3-37
link_f 3-40
lock_f 3-41
lseek_f 3-43
lstat_f 3-45
make_dir 3-47
mount_vol 3-49
move_f 3-51
nfsmount_vol 3-53
open_dir 3-55
open_f 3-56
open_fn 3-59
pcinit_vol 3-61
pcmount_vol 3-65
read_dir 3-67
read_f 3-69
read_link 3-71
read_vol 3-73
remove_f 3-75
stat_f 3-76
stat_vfs 3-80
symlink_f 3-83
sync_vol 3-84
truncate_f 3-86
unmount_vol 3-88
utime_f 3-90

verify_vol 3-91
write_f 3-107
write_vol 3-109

4 pREPC+ System Calls

abort 4-9
abs 4-10
acos 4-11
asctime 4-12
asctime_r 4-14
asin 4-16
assert 4-17
atan 4-19
atan2 4-20
atexit 4-22
atof 4-23
atoi 4-25
atol 4-27
bsearch 4-29
calloc 4-31
ceil 4-33
clearerr 4-34
cos 4-35
cosh 4-36
ctime 4-37
ctime_r 4-39
difftime 4-41

div 4-42
errno 4-44
exit 4-45
exp 4-47
fabs 4-48
fclose 4-49
feof 4-51
ferror 4-52
fflush 4-53
fgetc 4-55
fgetpos 4-56
fgets 4-57
floor 4-59
fmod 4-60
fopen 4-62
fprintf 4-66
fputc 4-71
fputs 4-73
fread 4-75
free 4-77
freopen 4-79
frexp 4-81
fscanf 4-83
fseek 4-87
fsetpos 4-89
ftell 4-91
fwrite 4-93

getc 4-95
getchar 4-96
gets 4-97
gmtime 4-99
gmtime_r 4-101
isalnum 4-103
isalpha 4-105
iscntrl 4-107
isdigit 4-109
isgraph 4-111
islower 4-113
isprint 4-115
ispunct 4-117
isspace 4-119
isupper 4-121
isxdigit 4-123
labs 4-125
ldexp 4-126
ldiv 4-128
localeconv 4-130
localtime 4-133
localtime_r 4-135
log 4-137
log10 4-138
longjmp 4-139
malloc 4-141
mblen 4-142

mbstowcs 4-144
mbtowc 4-146
memccpy 4-148
memchr 4-150
memcmp 4-152
memcpy 4-154
memmove 4-156
memset 4-158
mktime 4-159
modf 4-162
perror 4-164
pow 4-165
printf 4-167
putc 4-169
putchar 4-171
puts 4-172
qsort 4-173
rand 4-175
rand_r 4-176
realloc 4-178
remove 4-180
rename 4-181
rewind 4-183
scanf 4-185
setbuf 4-187
setjmp 4-189
setlocale 4-191

setvbuf 4-194
sin 4-196
sinh 4-197
sprintf 4-198
sqrt 4-200
srand 4-201
sscanf 4-203
strcat 4-205
strchr 4-206
strcmp 4-207
strcoll 4-208
strcpy 4-210
strcspn 4-211
strerror 4-212
strftime 4-214
strlen 4-217
strncat 4-218
strncmp 4-220
strncpy 4-222
strpbrk 4-224
strrchr 4-225
strspn 4-226
strstr 4-227
strtod 4-228
strtok 4-230
strtok_r 4-232
strtol 4-234

strtoul 4-236
strxfrm 4-238
tan 4-240
tanh 4-241
time 4-242
tmpfile 4-244
tmpnam 4-245
tolower 4-247
toupper 4-249
ungetc 4-251
vfprintf 4-253
vprintf 4-255
vsprintf 4-257
wcstombs 4-259
wctomb 4-261

5 pLM+ System Calls

sl_acquire 5-3
sl_bindindex 5-10
sl_getattr 5-11
sl_getindex 5-13
sl_getsymaddr 5-15
sl_register 5-17
sl_release 5-23
sl_setattr 5-28
sl_unregister 5-30
sl_update 5-34

6 pNA+ System Calls

accept 6-5
add_ni 6-7
bind 6-9
close 6-11
connect 6-12
get_id 6-14
getpeername 6-15
getsockname 6-17
getsockopt 6-19
ioctl 6-23
listen 6-41
pna_allocb 6-42
pna_esballoc 6-44
pna_freeb 6-46
pna_freemsg 6-47
recv 6-48
recvfrom 6-51
recvmsg 6-54
select 6-57
send 6-60
sendmsg 6-62
sendto 6-64
set_id 6-67
setsockopt 6-68
shr_socket 6-75

shutdown 6-76
socket 6-77

7 pRPC+ System Calls

clnt_control 7-5
get_fdset 7-7
rpc_getcreateerr 7-8

8 pROBE+ and ESp System Calls

db_input 8-3
db_output 8-5
log_event 8-6

A Error Codes

A.1 pSOS+ Error Codes A-4
A.2 pROBE+ Error Codes A-19
A.3 pHILE+ Error Codes A-20
A.3.1 pSOS+ Errors Related to pHILE+ A-41
A.3.2 Conversions of NFS Error Codes A-41
A.3.3 Conversions of RPC Error Codes A-43
A.4 pREPC+ Error Codes A-45
A.5 pLM+ Error Codes A-46
A.6 pNA+ Error Codes A-49
A.7 pRPC+ Error Codes A-55
A.8 pMONT+ Error Codes A-57
A.9 Driver Error Codes A-58
A.9.1 Shared Memory Network Interface Driver Error Codes A-58

A.9.2 Shared Memory Kernel Interface Driver Error Codes A-59
A.9.3 Terminal Interface Driver Error Codes A-59
A.9.4 Tick Timer Driver Error Codes A-61
A.9.5 RAM Disk Driver Error Codes A-61
A.9.6 TFTP Driver Error Codes A-61
A.9.7 IDE Driver Error Codes A-62
A.9.8 FLP Driver Error Codes A-63
A.9.9 HTTP Driver Error Codes A-64
A.9.10 Pipe Driver Error Codes A-64
A.9.11 RDIO Driver Error Codes A-65
A.9.12 LAN Driver Error Codes A-65
A.9.13 SCSI Driver Error Codes A-66
A.9.14 Parallel Driver Error Codes A-68

Index **index-1**

Using This Manual

This manual is part of a documentation set that describes pSOSystem, the modular, high-performance real-time operating system environment from Integrated Systems, Inc. This manual is targeted for embedded application developers using the pSOSystem environment. Basic familiarity with UNIX terms and concepts is assumed.

System Calls contains detailed descriptions of all pSOSystem system calls and error codes. For purpose of usability, *System Calls* provides an appendix that provides a numerical list of all error codes returned by pSOSystem. Each error code is listed with its description and the system calls that can return it.

The basic pSOSystem documentation set consists of *pSOSystem System Calls*, *User's Guide*, *pSOSystem System Concepts*, *pSOSystem Programmer's Reference*, *pSOSystem Advanced Topics*, and *pSOSystem Application Examples*.

Organization

This manual is organized as follows:

Chapter 1, *pSOSystem System Calls*, summarizes all system calls in pSOSystem.

Chapter 2, *pSOS+ System Calls*, details each system call in the pSOS+/pSOS+m component of pSOSystem.

Chapter 3, *pHILE+ System Calls*, details each system call in the pHILE+ component of pSOSystem.

Chapter 4, *pREPC+ System Calls*, details each system call in the pREPC+ component of pSOSystem.

Chapter 5, *pLM+ System Calls*, details each system call in the pLM+ component of pSOSystem.

Chapter 6, *pNA+ System Calls*, details each system call in the pNA+ component of pSOSystem.

Chapter 7, *pRPC+ System Calls*, details each system call in the pRPC+ component of pSOSystem.

Chapter 8, *pROBE+ and ESp System Calls*, details the system calls supported by the pROBE+ target debugger/analyzer and the ESp cross-system visual analyzer.

Appendix A, *Error Codes*, lists all error codes returned by pSOSystem system calls.

The *Index* provides a quick way to find a system call. If you know the name of the call, the function of the call, or the name of a parameter in the call, you have a good chance of locating the call with the index.

Conventions

This section describes the conventions used in this manual.

Font Conventions

Fonts other than the standard text default font are used as follows:

`Courier`	`Courier` is used for command and function names, file names, directory paths, environment variables, messages and other system output, code and program examples, system calls, and syntax examples.
`bold Courier`	User input (anything you are expected to type in) is set in **`bold Courier`**.
italic	*Italic* is used in conjunction with the default font for emphasis, first instances of terms defined in the glossary, and publication titles.
Bold Helvetica narrow	Buttons, fields, and icons in a graphical user interface are set in **bold Helvetica narrow** type. Keyboard keys are also set in this type.

Symbol Conventions

This section describes symbol conventions used in this document.

[ ]	Brackets indicate that the enclosed information is optional. The brackets are generally not typed when the information is entered.
|	A vertical bar separating two text items indicates that either item can be entered as a value.
˘	The breve symbol indicates a required space (for example, in user input).
%	The percent sign indicates the UNIX operating system prompt for C shell.
$	The dollar sign indicates the UNIX operating system prompt for Bourne and Korn shells.
	The symbol of a processor located to the left of text identifies processor-specific information (the example identifies 68K-specific information).
	Host tool-specific information is identified by a host tools icon (in this example, the text would be specific to the XXXXX host tools chain).

Format Conventions

Each reference section in this manual adheres to a standard format. The name of the system call, a brief description, and its C language syntax appear at the top of the first page. The remaining information about the call appears below the syntax and is organized under the following headings:

Volume Types

For pHILE+ system calls only. Names the volume types the system call supports. Volume types include pHILE+, NFS, MS-DOS, and CD-ROM.

Description

Provides a description of the call.

Arguments

Provides descriptions of all arguments used in the call.

Target

Where applicable, provides processor-specific information about the call. The information appears next to an icon representing the processor in question, as below:

On 68K processors, a signal is passed to the ASR in the D0.L register.

If the information is also specific to a set of host tools, a host tool icon appears next to the processor icon, as below:

For 68K processors with XXXXX host tools, the formula is the following:

SIZE = 32 + (4 * VSIZE) + (16 * NFD) + (42 * MAXDEPTH)

Return Value

Lists the possible return values of the call. For example, pSOS+ system calls always return a 0 to indicate a successful call, and pREPC+ calls can return either a non-zero value if the test result is true, or 0 if it is false. The Error Codes section lists possible errors returned by each call.

Error Codes

Provides a list of the error codes that the call can generate. For pSOS+ and pHILE+ system calls, an error code returns as a system call *return value*. For other components, such as the pREPC+ component/library and the pNA+ network manager, error codes are loaded into an internal variable that can be read through the macro `errno`. Appendix A contains a complete list of error codes for each software component.

Usage

Provides detailed usage information for certain system calls. For instance, the `verify_vol()` call of the pHILE+ component performs multiple actions that require detailed explanations.

Notes

Provides supplemental information, warnings, and side effects of a call. For pSOS+ system calls, a subsection called “Multiprocessor Considerations” describes the behavior of the call in a multiprocessing environment if it differs from that in a single-processor environment. Also, the subsection “Callable From” lists classes of

program elements that the system call can be called from. The system will deadlock if the call is made from a program element not listed in the "Callable From" subsection. There are four possible program elements:

Task — The smallest unit of execution that can compete on its own for system resources.

ISR — Interrupt Service Routine. A function that takes control of the system when the CPU has been triggered with an exception from an external source. An ISR is part of a device driver.

KI — Kernel Interface. The kernel interface is used by pSOS+m to communicate with other pSOS+m kernels on other processors.

Callout — A function that a device driver uses to notify a pSOSystem component of an interrupt event. A callout is called from an ISR.

See Also

Lists related service calls or the location of other relevant information.

Note, Caution, and Warning Conventions

Within the text of this manual, you may find notes, cautions, and warnings. These statements are used for the purposes described below.

NOTE: Notes provide special considerations or details which are important to the procedures or explanations presented.

CAUTION: **Cautions indicate actions that may result in possible loss of work performed and associated data. An example might be a system crash that results in the loss of data for that given session.**

WARNING: **Warnings indicate actions or circumstances that may result in file corruption, irrecoverable data loss, data security risk, or damage to hardware.**

Revision Bar Convention

A *revision bar*, at left, appears in the margin next to any text that has changed since the last release of this manual.

Commonly Used Terms and Acronyms

The following terms and acronyms are commonly associated with pSOSystem and appear in this manual.

ASR	*See* asynchronous signal routine.
asynchronous signal routine	A function within an application that executes in response to an asynchronous signal.
callout	A function that a device driver uses to notify a pSOSystem component of an interrupt event. A callout is called from an ISR.
FD	File descriptor.
FLIST	A contiguous sequence of blocks used to hold file descriptors on a pHILE+ formatted volume.
ISR	*See* interrupt service routine.
interrupt service routine	A function within an application or device driver that takes control of the system when the CPU has been triggered with an exception from an external source.
KI	*See* kernel interface.
kernel interface	A user-provided communication layer between nodes in a multiprocessing environment (pSOS+m).
NFS	Network file system.
NI	Network interface.
RSC	*See* remote service call.
remote service call	A service call made from one node to another in a multiprocessing environment (pSOS+m).
ROOTBLOCK	The root block on a pHILE+ formatted volume, which contains all information needed by pHILE+ to locate other vital information on the volume.
socket	The endpoint for communication across a network.
task	The smallest unit of execution in a system designed with pSOSystem that can compete on its own for system resources.
TCP/IP	Transport Control Protocol/Internet Protocol, a software protocol for communications between computers.
UDP	User Datagram Protocol.

Related Publications

When using the pSOSystem operating system, you might want to have on hand the other manuals included in the basic documentation set:

- *User's Guide* contains both introductory and detailed information about using pSOSystem. The introductory material includes tutorials, a description of board-support packages, configuration instructions, information on files and directories, board-specific information, and using the pROBE+ debugger. The rest of the manual provides detailed information for more advanced users.
- *pSOSystem System Concepts* describes the operation of pSOSystem.
- *pSOSystem Programmer's Reference* is the primary source of information on network drivers and interfaces, system services, configuration tables, memory-usage data, and processor-specific assembly languages.
- *pSOSystem Advanced Topics* describes processor-specific assembly language information, and also describes how to develop Board-Support Packages.
- *pSOSystem Application Examples* and *pSOSystem Supplemental Application Examples on the Integrated Systems' Web Site* provide information on how to access, build, and execute the pSOSystem application examples.
- *pROBE+ User's Guide* tells how to use the pROBE+ target debugger/analyzer.

Based on the options you purchased, you may need some of the following manuals:

- *CD-ROM Installation for Windows* describes how to install your system on Windows.
- *CD-ROM Installation for UNIX* describes how to install your system on UNIX.
- *Using this Documentation CD-ROM* describes how to use the documentation CD-ROM.
- *C++ Support Package User's Guide* documents the C++ support services including the pSOSystem C++ Classes (library) and support for the C++ run time.
- *OpEN User's Guide* describes how to install and use pSOSystem's OpEN (Open Protocol Embedded Networking) product.
- *SNMP User's Guide* describes the internal structure and operation of SNMP (Integrated System's Simple Network Management Protocol product), and also how to install and use the SNMP MIB (Management Information Base) compiler.

- The *Upgrade Guide* describes how to upgrade your system to the current release level.
- The *System Administration Guide: License Manager* describes how to complete your software installation by installing a permanent license for your software.
- *RTA Suite Visual Run-Time Analysis Tools User's Guide* describes how to use the run-time analysis tools.
- *POSIX Support Package User's Guide* describes how to use the POSIX support package.
- *TCP/IP for OpEN User's Guide* describes how to use the pSOSystem Streams-based TCP/IP for OpEN (Open Protocol Embedded Networking) product.
- *Point-to-Point Protocol Driver User's Guide* describes how to use the point-to-point protocol, which is a data link layer protocol that encapsulates multiple network layer packets to run over a serial connection.
- *X.25 for OpEN User's Guide* describes the interfaces provided by the X.25 for the OpEN multiplexing driver that implements the packet level protocol.
- *LAP Driver User's Guide* describes the interfaces provided by the LAP (Link Access Protocol) drivers for OpEN product, including the LAPB and LABD frame-level products.
- *Using pNAT* describes how to use the pNAT component of pSOSystem.

The following non-Integrated Systems' documentation might also be needed:

- *Diab Data version 4.2 documentation set* (part number 018-5001-001).
- *Using Diab Data with pSOS*.
- *SDS version 7.4* (part number 000-5423-001).

Support

Customers in the United States can contact Integrated Systems Technical Support as described below.

International customers can contact:

- The local Integrated Systems branch office.
- The local pSOSystem distributor.
- Integrated Systems Technical Support as described below.

Before contacting Integrated Systems Technical Support, please gather the following information available:

- Your customer ID and complete company address.
- Your phone and fax numbers and e-mail address.
- Your product name, including components, and the following information:
  - The version number of the product.
  - The host and target systems.
  - The type of communication used (Ethernet, serial).
- Your problem (a brief description) and the impact to you.

In addition, please gather the following information:

- The procedure you followed to build the code. Include components used by the application.
- A complete record of any error messages as seen on the screen (useful for tracking problems by error code).
- A complete test case, if applicable. Attach all include or startup files, as well as a sequence of commands that will reproduce the problem.

Contacting Integrated Systems Support

To contact Integrated Systems Technical Support, use one of the following methods:

- Call 408-542-1925 (U.S. and international countries).
- Call 1-800-458-7767 (1-800-458-pSOS) (U.S. and international countries with 1-800 support).
- Send a FAX to 408-542-1966.
- Send e-mail to `psos_support@isi.com`.
- Access our web site: `http://customer.isi.com`.

Integrated Systems actively seeks suggestions and comments about our software, documentation, customer support, and training. Please send your comments by e-mail to `ideas@isi.com` or submit a Problem Report form via the internet (http://customer.isi.com/report.shtml).

1 pSOSystem System Calls

This manual provides detailed information on each system call in all of the components of pSOSystem.

This chapter provides a table (Table 1-1 on page 1-2) that lists all pSOSystem system calls alphabetically, and provides for each call a one-line description, the pSOSystem component it belongs to, and the page number where you can find more information.

The rest of the chapters in this manual each describe the calls for one component of pSOSystem. Each chapter has a brief introduction followed by a table that alphabetically lists the system calls for that component (i.e., pSOS+, pHILE+, etc.) and provides for each call a one-line description and the page number where you can find more information.

Appendix A, *Error Codes*, provides error code information for all of the components of pSOSystem.

TABLE 1-1 All pSOSystem System Calls

Name	Component	Description	Page
abort	pREPC+	Aborts a task.	4-9
abs	pREPC+	Computes the absolute value of an integer	4-10
accept	pNA+	Accepts a connection on a socket.	6-5
access_f	pHILE+	Determines the accessibility of a file.	3-7
add_ni	pNA+	Adds a network interface.	6-7
annex_f	pHILE+	Allocates contiguous blocks to a file.	3-8
as_catch	pSOS+	Specifies an asynchronous signal routine.	2-9
asctime	pREPC+	Converts the broken-down time to a string.	4-12
asctime_r	pREPC+	(Reentrant) Converts the broken-down time to a string.	4-14
as_notify	pSOS+	Registers events for notification of a signal.	2-14
as_return	pSOS+	Returns from an asynchronous signal routine.	2-16
as_send	pSOS+	Sends asynchronous signals to a task.	2-18
assert	pREPC+	Verifies that a program is operating correctly.	4-17
atexit	pREPC+	Registers functions.	4-22
atof	pREPC+	Converts a string to a double.	4-23
atoi	pREPC+	Converts a string to an integer.	4-25
atol	pREPC+	Converts a string to a long integer.	4-27
bind	pNA+	Binds an address to a socket.	6-9
bsearch	pREPC+	Searches an array.	4-29
calloc	pREPC+	Allocates memory.	4-31
cdmount_vol	pHILE+	Mounts a CD-ROM volume.	3-10
change_dir	pHILE+	Changes the current directory.	3-12
chmod_f	pHILE+	Changes the mode of a named ordinary or directory file.	3-14

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
chown_f	pHILE+	Changes the owner or group of a named ordinary or directory file.	3-16
clearerr	pREPC+	Clears a stream's error indicators.	4-34
clnt_control	pRPC+	Change the internal client creation timeout values.	7-5
close	pNA+	Closes a socket descriptor.	6-11
close_dir	pHILE+	Closes an open directory file.	3-17
close_f	pHILE+	Closes an open file connection.	3-17
connect	pNA+	Initiates a connection on a socket.	6-12
control_vol	pHILE+	Performs control operations on a volume.	3-20
co_register	pSOS+	Registers a callout handler.	2-20
co_unregister	pSOS+	Un-registers a callout handler.	2-23
create_f	pHILE+	Creates a data file.	3-23
ctime	pREPC+	Converts the calendar time to a string.	4-37
ctime_r	pREPC+	(Reentrant) Converts the calendar time to a string.	4-39
cv_abroadcast	pSOS+	Asynchronously signals all the tasks waiting on a condition variable to run.	2-25
cv_asignal	pSOS+	Sends asynchronous signals to a task waiting on a condition variable.	2-27
cv_broadcast	pSOS+	Signals all the tasks waiting on a condition variable to run.	2-29
cv_create	pSOS+	Creates a condition variable.	2-31
cv_delete	pSOS+	Deletes a condition variable.	2-33
cv_ident	pSOS+	Obtains the identifier of a named condition variable.	2-35
cv_info	pSOS+	Query about a Condition Variable Object.	2-37

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
cv_signal	pSOS+	Signals a task waiting on a condition variable to run.	2-39
cv_wait	pSOS+	Waits on a condition variable.	2-41
db_input	pROBE+	Prompts and gets input from the high-level debugger.	8-3
db_output	pROBE+	Outputs a string to the high-level debugger.	8-5
de_close	pSOS+	Closes an I/O device.	2-43
de_cntrl	pSOS+	Requests a special I/O device service.	2-45
de_init	pSOS+	Initializes an I/O device and its driver.	2-47
de_open	pSOS+	Opens an I/O device.	2-49
de_read	pSOS+	Reads data from an I/O device.	2-51
de_write	pSOS+	Writes data to an I/O device.	2-53
difftime	pREPC+	Computes the difference between two calendar times.	4-41
div	pREPC+	Performs a division operation on two specified integers.	4-42
dnt_add	pSOS+	Adds a new device name to the DNT.	2-55
dnt_remove	pSOS+	Removes a device name from the DNT.	2-59
dnt_find	pSOS+	Returns the major/minor number associated with a given device name.	2-57
errno	pREPC+	The error number returned by the last failing system call.	4-44
errno_addr	pSOS+	Obtains the address of the calling task's internal errno variable.	2-60
ev_asend	pSOS+	(pSOS+m kernel only) Asynchronously sends events to a task.	2-62
ev_receive	pSOS+	Allows a task to wait for an event condition.	2-64
ev_send	pSOS+	Sends events to a task.	2-67

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
exit	pREPC+	Terminates a task.	4-45
fchmod_f	pHILE+	Changes the mode of an ordinary or directory file specified by its file identifier.	3-25
fchown_f	pHILE+	Changes the owner or group of a file specified by its file identifier.	3-27
fclose	pREPC+	Closes a stream.	4-49
feof	pREPC+	Tests a stream's end-of-file indicator.	4-51
ferror	pREPC+	Tests a stream's error indicator.	4-52
fflush	pREPC+	Flushes the buffer associated with an open stream.	4-53
fgetc	pREPC+	Gets a character from a stream.	4-55
fgetpos	pREPC+	Gets the current file position indicator for fsetpos.	4-56
fgets	pREPC+	Gets a string from a stream.	4-57
fopen	pREPC+	Opens a file.	4-62
fprintf	pREPC+	Prints formatted output to a stream.	4-66
fputc	pREPC+	Writes a character to a stream.	4-71
fputs	pREPC+	Writes a string to a stream.	4-73
fread	pREPC+	Reads from a stream.	4-75
free	pREPC+	Deallocates memory.	4-77
freopen	pREPC+	Reopens a file.	4-79
fscanf	pREPC+	Reads formatted input from a stream.	4-83
fseek	pREPC+	Sets the file position indicator.	4-87
fsetpos	pREPC+	Sets file position by using the fgetpos result.	4-89
fstat_f	pHILE+	Obtains the status of a file specified by its file identifier.	3-28

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
fstat_vfs	pHILE+	Obtains statistics about a mounted volume specified by a file identifier.	3-31
ftell	pREPC+	Gets the file position indicator.	4-91
ftruncate_f	pHILE+	Changes the size of a file specified by its file identifier.	3-33
fwrite	pREPC+	Writes to a stream.	4-93
get_fdset	pRPC+	Returns the bit mask that corresponds to readable RPC sockets.	7-7
get_fn	pHILE+	Obtains the file number of a file.	3-35
get_id	pNA+	Gets a task's user ID and group ID.	6-14
getc	pREPC+	Gets a character from a stream.	4-95
getchar	pREPC+	Gets a character from stdin.	4-96
getpeername	pNA+	Gets the address of a connected peer.	6-15
gets	pREPC+	Gets a string from stdin.	4-97
getsockname	pNA+	Gets the address that is bound to a socket.	6-17
getsockopt	pNA+	Gets options on a socket.	6-19
gmtime	pREPC+	Converts the calendar time to broken-down time.	4-99
gmtime_r	pREPC+	(Reentrant) Converts the calendar time to broken-down time.	4-101
init_vol	pHILE+	Initializes a pHILE+ formatted volume.	3-37
ioctl	pNA+	Performs control operations on a socket.	6-23
ioj_bind	pSOS+	Binds a particular I/O Jump Table entry and a driver.	2-69
ioj_bindany	pSOS+	Binds any I/O Jump Table entry to a driver.	2-71
ioj_getent	pSOS+	Obtain information about a particular I/O Jump Table entry.	2-73

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
ioj_lock	pSOS+	Increments the lock count of an I/O Jump Table entry.	2-75
ioj_unbind	pSOS+	Unbinds an I/O Jump Table entry and a driver.	2-76
ioj_unlock	pSOS+	Decrements the lock count of an I/O Jump Table entry.	2-78
isalnum	pREPC+	Tests for an alphanumeric character.	4-103
isalpha	pREPC+	Tests for an alphabetic character.	4-105
iscntrl	pREPC+	Tests for a control character.	4-107
isdigit	pREPC+	Tests for a digit.	4-109
isgraph	pREPC+	Tests for a graphical character.	4-111
islower	pREPC+	Tests for a lowercase letter.	4-113
isprint	pREPC+	Tests for a printable character.	4-115
ispunct	pREPC+	Tests for a punctuation character.	4-117
isspace	pREPC+	Tests for a space.	4-119
isupper	pREPC+	Tests for an uppercase letter.	4-121
isxdigit	pREPC+	Tests for a hexadecimal digit.	4-123
k_fatal	pSOS+	Aborts and enters fatal error handling mode.	2-79
k_terminate	pSOS+	Terminates a node other than the master node.	2-81
labs	pREPC+	Computes the absolute value of a long integer.	4-125
ldiv	pREPC+	Performs a division operation on two specified long integers.	4-128
link_f	pHILE+	Creates a hard link between two files on the same volume.	3-40
listen	pNA+	Listens for connections on a socket.	6-41

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
localeconv	pREPC+	Obtains the current locale settings.	4-130
localtime	pREPC+	Converts the calendar time to broken-down time.	4-133
localtime_r	pREPC+	(Reentrant) Converts the calendar time to broken-down time.	4-135
lock_f	pHILE+	Locks or unlocks part or all of an open file.	3-41
log_event	ESp	Logs an event on ESp's target-resident application monitor, pMONT.	8-6
longjmp	pREPC+	Restores the environment set by the most recent invocation of setjmp.	4-139
lseek_f	pHILE+	Repositions for read or write within an open file.	3-43
lstat_f	pHILE+	Gets the status of a symbolically linked file.	3-45
m_ext2int	pSOS+	Converts an external address into an internal address.	2-83
m_int2ext	pSOS+	Converts an internal address into an external address.	2-85
make_dir	pHILE+	Creates a directory file.	3-47
malloc	pREPC+	Allocates memory.	4-141
mblen	pREPC+	Determines the number of bytes in a multibyte character.	4-142
mbstowcs	pREPC+	Converts a multibyte character string into a wide character string.	4-144
mbtowc	pREPC+	Converts a multibyte character into its wide character equivalent.	4-146
memccpy	pREPC+	Copies characters from one memory area to another.	4-148
memchr	pREPC+	Searches memory for a character.	4-150
memcmp	pREPC+	Compares two objects in memory.	4-152

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
memcpy	pREPC+	Copies characters in memory.	4-154
memmove	pREPC+	Copies characters in memory.	4-156
memset	pREPC+	Initializes a memory area with a given value.	4-158
mktime	pREPC+	Converts the broken-down time into calendar time.	4-159
mount_vol	pHILE+	Mounts a pHILE+ formatted volume.	3-49
move_f	pHILE+	Moves (renames) a file.	3-51
mu_create	pSOS+	Creates a mutex.	2-87
mu_delete	pSOS+	Deletes a mutex.	2-90
mu_ident	pSOS+	Obtains the identifier of a named mutex.	2-92
mu_info	pSOS+	Query about a Mutex Object	2-94
mu_lock	pSOS+	Locks a mutex.	2-96
mu_unlock	pSOS+	Unlocks a mutex.	2-100
nfsmount_vol	pHILE+	Mounts a remote file system.	3-53
ob_roster	pSOS+	Obtains a roster of pSOS+ objects according to specified type.	2-102
open_dir	pHILE+	Opens a directory file.	3-55
open_f	pHILE+	Opens a file.	3-56
open_fn	pHILE+	Opens a file by its file identifier.	3-59
pcinit_vol	pHILE+	Initializes an MS-DOS volume.	3-61
pcmount_vol	pHILE+	Mounts an MS-DOS volume.	3-65
perror	pREPC+	Prints a diagnostic message.	4-164
pna_allocb	pNA+	Allocates a message block.	6-42
pna_esballoc	pNA+	Attaches a message block to the data buffer.	6-44
pna_freeb	pNA+	Frees a message block.	6-46

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
pna_freemsg	pNA+	Frees all the message blocks associated with a message.	6-47
printf	pREPC+	Prints formatted output to stdout.	4-167
pt_create	pSOS+	Creates a memory partition of fixed-size buffers.	2-105
pt_delete	pSOS+	Deletes a memory partition.	2-108
pt_getbuf	pSOS+	Gets a buffer from a partition.	2-110
pt_ident	pSOS+	Obtains the identifier of the named partition.	2-112
pt_info	pSOS+	Queries a Partition Object.	2-114
pt_retbuf	pSOS+	Returns a buffer to the partition from which it came.	2-116
pt_sgetbuf	pSOS+	Gets a buffer from a partition.	2-118
putc	pREPC+	Writes a character to a stream.	4-169
putchar	pREPC+	Writes a character to stdout.	4-171
puts	pREPC+	Writes a string to a file.	4-172
q_asend	pSOS+	(pSOS+m kernel only) Asynchronously posts a message to an ordinary message queue.	2-120
q_aurgent	pSOS+	(pSOS+m kernel only) Asynchronously posts a message at the head of an ordinary message queue.	2-122
q_avsend	pSOS+	(pSOS+m kernel only) Asynchronously posts a message to a variable-length message queue.	2-124
q_avurgent	pSOS+	(pSOS+m kernel only) Asynchronously posts a message at the head of a variable-length message queue.	2-126
q_broadcast	pSOS+	Broadcasts identical messages to an ordinary message queue.	2-128
q_create	pSOS+	Creates an ordinary message queue.	2-130

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
q_delete	pSOS+	Deletes an ordinary message queue.	2-133
q_ident	pSOS+	Obtains the queue ID of an ordinary message queue.	2-135
q_info	pSOS+	Queries a Message Queue Object.	2-137
q_notify	pSOS+	Registers the events for notification of availability of a message.	2-139
q_receive	pSOS+	Requests a message from an ordinary message queue.	2-141
q_send	pSOS+	Posts a message to an ordinary message queue.	2-144
q_urgent	pSOS+	Posts a message to the head of an ordinary message queue.	2-146
q_vbroadcast	pSOS+	Broadcasts identical variable-length messages to a variable-length message queue.	2-148
q_vcreate	pSOS+	Creates a variable-length message queue.	2-150
q_vdelete	pSOS+	Deletes a variable-length message queue.	2-153
q_vident	pSOS+	Obtains the queue ID of a variable-length message queue.	2-155
q_vinfo	pSOS+	Queries a Variable Length Message Queue Object.	2-157
q_vnotify	pSOS+	Registers the events for notification of availability of a message.	2-159
q_vreceive	pSOS+	Requests a message from a variable-length message queue.	2-161
q_vsend	pSOS+	Posts a message to a specified variable-length message queue.	2-163
q_vurgent	pSOS+	Posts a message at the head of a variable-length message queue.	2-165
qsort	pREPC+	Sorts an array in ascending order.	4-173

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
rand	pREPC+	Returns a pseudo-random number.	4-175
rand_r	pREPC+	Returns a pseudo-random sequence of integers with repeated calls.	4-176
read_dir	pHILE+	Reads directory entries in a file system independent format.	3-67
read_f	pHILE+	Reads from a file.	3-69
read_link	pHILE+	Reads the value of a symbolic link.	3-71
read_vol	pHILE+	Reads directly from a pHILE+ formatted volume.	3-73
realloc	pREPC+	Allocates memory.	4-178
recv	pNA+	Receives data from a socket.	6-48
recvfrom	pNA+	Receives data from a socket.	6-51
recvmsg	pNA+	Receives data from a socket.	6-54
remove	pREPC+	Removes a file.	4-180
remove_f	pHILE+	Deletes a file.	3-75
rename	pREPC+	Renames a file.	4-181
rewind	pREPC+	Resets the file position indicator.	4-183
rn_create	pSOS+	Creates a memory region.	2-167
rn_delete	pSOS+	Deletes a memory region.	2-170
rn_getseg	pSOS+	Allocates a memory segment to the calling task.	2-172
rn_ident	pSOS+	Obtains the region identifier of the named region.	2-174
rn_info	pSOS+	Queries a Region Object.	2-176
rn_retseg	pSOS+	Returns a memory segment to the region from which it was allocated.	2-178

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
rpc_getcreateerr	pRPC+	Returns the reason for an RPC client handle creation failure.	7-8
scanf	pREPC+	Reads formatted input from stdin.	4-185
select	pNA+	Checks the status of multiple sockets.	6-57
send	pNA+	Sends data to a socket.	6-60
sendmsg	pNA+	Sends data to a socket.	6-62
sendto	pNA+	Sends data to a socket.	6-64
set_id	pNA+	Sets a task's user ID and group ID.	6-67
setbuf	pREPC+	Changes a stream's buffer.	4-187
setjmp	pREPC+	Saves the calling environment for later restoration.	4-189
setlocale	pREPC+	Obtains or changes the program's locale.	4-191
setsockopt	pNA+	Sets options on a socket.	6-68
setvbuf	pREPC+	Changes a stream's buffering characteristics.	4-194
shr_socket	pNA+	Obtains a new socket descriptor for an existing socket.	6-75
shutdown	pNA+	Terminates all or part of a full-duplex connection.	6-76
sl_acquire	pLM+	Acquires a shared library for the calling process.	5-3
sl_bindindex	pLM+	Gets the library's index for the data section variable in the stub file.	5-10
sl_getattr	pLM+	Obtains the attributes of a registered library.	5-11
sl_getindex	pLM+	Obtains the index of a registered shared library.	5-13
sl_getsymaddr	pLM+	Obtains the address of the symbol defined within a registered library,	5-15
sl_register	pLM+	Adds a shared library to the pLM+ table.	5-17

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
sl_release	pLM+	Releases a previously acquired shared library index by the calling process.	5-23
sl_setattr	pLM+	Sets the writable attributes of a registered library.	5-28
sl_unregister	pLM+	Unregisters a shared library from the pLM+ table.	5-30
sl_update	pLM+	Replaces a currently registered library with another library with the same name.	5-34
sm_av	pSOS+	(pSOS+m kernel only) Asynchronously releases a semaphore token.	2-180
sm_create	pSOS+	Creates a semaphore.	2-182
sm_delete	pSOS+	Deletes a semaphore.	2-185
sm_ident	pSOS+	Obtains a semaphore identifier.	2-187
sm_info	pSOS+	Queries a Semaphore Object.	2-189
sm_notify	pSOS+	Registers the events for notification of semaphore availability.	2-191
sm_p	pSOS+	Acquires a semaphore token.	2-193
sm_v	pSOS+	Releases a semaphore token.	2-195
socket	pNA+	Creates a socket.	6-77
sprintf	pREPC+	Writes formatted output to a buffer.	4-198
srand	pREPC+	Sets the seed for the random number generator (rand).	4-201
sscanf	pREPC+	Reads formatted input from a string.	4-203
stat_f	pHILE+	Gets the status of a named file.	3-76
stat_vfs	pHILE+	Gets statistics for a named volume.	3-80
strcat	pREPC+	Appends one string to another string.	4-205
strchr	pREPC+	Searches a string for a character.	4-206

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
strcmp	pREPC+	Compares two character strings.	4-207
strcoll	pREPC+	Compares two character strings.	4-208
strcpy	pREPC+	Copies one string to another string.	4-210
strcspn	pREPC+	Calculates the length of a substring.	4-211
strerror	pREPC+	Maps an error number to an error message string.	4-212
strftime	pREPC+	Places formatted time and date information into a string.	4-214
strlen	pREPC+	Computes string length.	4-217
strncat	pREPC+	Appends characters to a string.	4-218
strncmp	pREPC+	Compares characters in two strings.	4-220
strncpy	pREPC+	Copies characters from one string to another.	4-222
strpbrk	pREPC+	Searches a string for a character in a second string.	4-224
strrchr	pREPC+	Searches a string for a character.	4-225
strspn	pREPC+	Calculates specified string length.	4-226
strstr	pREPC+	Searches a string for specified characters in another string.	4-227
strtod	pREPC+	Converts a string to a double.	4-228
strtok	pREPC+	Searches a string for tokens.	4-230
strtok_r	pREPC+	Searches a string for tokens.	4-232
strtol	pREPC+	Converts a string to a long integer.	4-234
strtoul	pREPC+	Converts a string to an unsigned long.	4-236
strxfrm	pREPC+	Transforms a string so that it can be used by strcmp().	4-238
symlink_f	pHILE+	Creates a symbolic link to a file.	3-83

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
sync_vol	pHILE+	Synchronizes a volume.	3-84
sys_info	pSOS+	Obtains pSOS+ system information.	2-197
time	pREPC+	Obtains the current calendar time.	4-242
t_addvar	pSOS+	Adds a new task variable to a task.	2-201
t_create	pSOS+	Creates a task.	2-203
t_delete	pSOS+	Deletes a task.	2-208
t_delvar	pSOS+	Deletes a task variable.	2-211
t_getreg	pSOS+	Gets a task's notepad register.	2-213
t_ident	pSOS+	Obtains the task identifier of the named task.	2-215
t_info	pSOS+	Queries a Task Object.	2-217
t_mode	pSOS+	Gets or changes the calling task's execution mode.	2-221
t_restart	pSOS+	Forces a task to start over regardless of its current state.	2-225
t_resume	pSOS+	Resumes a suspended task.	2-227
t_setpri	pSOS+	Gets and optionally changes a task's priority.	2-229
t_setreg	pSOS+	Sets a task's notepad register.	2-231
t_start	pSOS+	Starts a task.	2-233
t_suspend	pSOS+	Suspends a task until a t_resume call is made for the suspended task.	2-237
t_tslice	pSOS+	Gets and optionally changes its own or another task's timeslice quantum.	2-239
tm_cancel	pSOS+	Cancels an armed timer.	2-241
tm_evafter	pSOS+	Sends events to the calling task at periodic intervals.	2-245
tm_evevery	pSOS+	Sends events to the calling task at periodic intervals.	2-245

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
tm_evwhen	pSOS+	Sends events to the calling task at the specified time.	2-247
tm_get	pSOS+	Obtains the system's current version of the date and time.	2-250
tm_getticks	pSOS+	Returns the elapsed tick count since the initialization of pSOS+.	2-252
tm_set	pSOS+	Sets or resets the system's version of the date and time.	2-254
tm_tick	pSOS+	Announces a clock tick to the pSOS+ kernel.	2-256
tm_wkafter	pSOS+	Blocks the calling task and wakes it after a specified interval.	2-258
tm_wkwhen	pSOS+	Blocks the calling task and wakes it at a specified time.	2-260
tmpfile	pREPC+	Creates a temporary file.	4-244
tmpnam	pREPC+	Generates a temporary filename.	4-245
tolower	pREPC+	Converts a character to lowercase.	4-247
toupper	pREPC+	Converts a character to uppercase.	4-249
truncate_f	pHILE+	Changes the size of a named file.	3-86
tsd_create	pSOS+	Creates a task-specific data (TSD) object.	2-262
tsd_delete	pSOS+	Deletes a task-specific data (TSD) object.	2-266
tsd_getval	pSOS+	Returns a pointer to the anchor for a task-specific data (TSD) access.	2-268
tsd_ident	pSOS+	Obtains the index for a named task-specific object.	2-270
tsd_info	pSOS+	Queries a Task-Specific Data Entry Object.	2-272
tsd_setval	pSOS+	Sets and gets a task's data value for a task-specific data (TSD).	2-274
ungetc	pREPC+	Ungets a character.	4-251

TABLE 1-1 All pSOSystem System Calls (Continued)

Name	Component	Description	Page
unmount_vol	pHILE+	Unmounts a volume.	3-88
utime_f	pHILE+	Sets the access and modification times of a file.	3-90
verify_vol	pHILE+	Verifies a volume's control structures.	3-91
vfprintf	pREPC+	Writes formatted output to a stream.	4-253
vprintf	pREPC+	Writes formatted output to stdout.	4-255
vsprintf	pREPC+	Writes formatted output to a buffer.	4-257
wcstombs	pREPC+	Converts a wide character string into a multibyte character string.	4-259
wctomb	pREPC+	Converts a wide character into its multibyte character equivalent.	4-261
write_f	pHILE+	Writes to an open file.	3-107
write_vol	pHILE+	Writes data directly to a pHILE+ formatted volume.	3-109

2 pSOS+ System Calls

This chapter provides detailed information on each system call in the pSOS+ component of pSOSystem. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, a detailed description, its arguments, its return value, and any error codes that it can return. In addition, it includes information specific to certain processors if such information is needed.

Where applicable, the section also includes the headings "Notes" and "See Also." The "Notes" entry provides important information not specifically related to the call description. In particular, it identifies how the pSOS+m kernel handles the call if multiple processors are involved (see "Multiprocessor Considerations,") and indicates where the call can be made. "See Also" lists other system calls that have related information.

If you need to look up a system call by its functionality, refer to Table 2-1 on page 2-2 which lists the calls alphabetically by component and provides a brief description of each call.

For more information on error codes, refer to Appendix A, *Error Codes*, which lists the codes numerically and shows which pSOSystem calls are associated with each one.

TABLE 2-1 pSOS+ System Calls

Name	Description	Page
as_catch	Specifies an asynchronous signal routine.	2-9
as_notify	Registers the events for notification of signal.	2-14
as_return	Returns from an asynchronous signal routine.	2-16
as_send	Sends asynchronous signals to a task.	2-18
co_register	Registers a callout handler.	2-20
co_unregister	Un-registers a callout handler.	2-23
cv_abroadcast	Asynchronously signals all the tasks waiting on a condition variable to run.	2-25
cv_asignal	Sends asynchronous signals to a task waiting on a condition variable.	2-27
cv_broadcast	Signals all the tasks waiting on a condition variable to run.	2-29
cv_create	Creates a condition variable.	2-31
cv_delete	Deletes a condition variable.	2-33
cv_ident	Obtains the identifier of a named condition variable.	2-35
cv_info	Queries about a Condition Variable Object.	2-37
cv_signal	Signals a task waiting on a condition variable to run.	2-39
cv_wait	Waits on a condition variable.	2-41
de_close	Closes an I/O device.	2-43
de_cntrl	Requests a special I/O device service.	2-45
de_init	Initializes an I/O device and its driver.	2-47
de_open	Opens an I/O device.	2-49
de_read	Reads data from an I/O device.	2-51
de_write	Writes data to an I/O device.	2-53
dnt_add	Adds a new device name to the DNT.	2-55
dnt_remove	Removes a device name from the DNT.	2-59

TABLE 2-1 pSOS+ System Calls (Continued)

Name	Description	Page
dnt_find	Returns the major/minor number associated with a given device name.	2-57
errno_addr	Obtains the address of the calling task's internal errno variable.	2-60
ev_asend	(pSOS+m kernel only) Asynchronously sends events to a task.	2-62
ev_receive	Allows a task to wait for an event condition.	2-64
ev_send	Sends events to a task.	2-67
ioj_bind	Binds a particular I/O Jump Table entry and a driver.	2-69
ioj_bindany	Binds any I/O Jump Table entry to a driver.	2-71
ioj_lock	Increments the lock count of an I/O Jump Table entry.	2-75
ioj_unbind	Unbinds an I/O Jump Table entry and a driver.	2-76
ioj_unlock	Decrements the lock count of an I/O Jump Table entry.	2-78
k_fatal	Aborts and enters fatal error handling mode.	2-79
k_terminate	Terminates a node other than the master node.	2-81
m_ext2int	Converts an external address into an internal address.	2-83
m_int2ext	Converts an internal address into an external address.	2-85
mu_create	Creates a mutex.	2-87
mu_delete	Deletes a mutex.	2-90
mu_ident	Obtains the identifier of a named mutex.	2-92
mu_info	Queries about a Mutex Object.	2-94
mu_lock	Locks a mutex.	2-96
mu_unlock	Unlocks a mutex.	2-100
ob_roster	Obtains a roster of pSOS+ objects according to a specified type.	2-102
pt_create	Creates a memory partition of fixed-size buffers.	2-105

TABLE 2-1 pSOS+ System Calls (Continued)

Name	Description	Page
pt_delete	Deletes a memory partition.	2-108
pt_getbuf	Gets a buffer from a partition.	2-110
pt_ident	Obtains the identifier of the named partition.	2-112
pt_info	Queries a Partition Object.	2-114
pt_retbuf	Returns a buffer to the partition from which it came.	2-116
pt_sgetbuf	Gets a buffer from a partition.	2-118
q_asend	(pSOS+m kernel only) Asynchronously posts a message to an ordinary message queue.	2-120
q_aurgent	(pSOS+m kernel only) Asynchronously posts a message at the head of an ordinary message queue.	2-122
q_avsend	(pSOS+m kernel only) Asynchronously posts a message to a variable-length message queue.	2-124
q_avurgent	(pSOS+m kernel only) Asynchronously posts a message at the head of a variable-length message queue.	2-126
q_broadcast	Broadcasts identical messages to an ordinary message queue.	2-128
q_create	Creates an ordinary message queue.	2-130
q_delete	Deletes an ordinary message queue.	2-133
q_ident	Obtains the queue ID of an ordinary message queue.	2-135
q_info	Queries a Message Queue Object.	2-137
q_notify	Registers the events for notification of availability of a message.	2-139
q_receive	Requests a message from an ordinary message queue.	2-141
q_send	Posts a message to an ordinary message queue.	2-144
q_urgent	Posts a message to the head of an ordinary message queue.	2-146
q_vbroadcast	Broadcasts identical variable-length messages to a variable-length message queue.	2-148

TABLE 2-1 pSOS+ System Calls (Continued)

Name	Description	Page
q_vcreate	Creates a variable-length message queue.	2-150
q_vdelete	Deletes a variable-length message queue.	2-153
q_vident	Obtains the queue ID of a variable-length message queue.	2-155
q_vinfo	Queries a Variable Length Message Queue Object.	2-157
q_vnotify	Registers the events for notification of availability of a message.	2-159
q_vreceive	Requests a message from a variable-length message queue.	2-161
q_vsend	Posts a message to a specified variable-length message queue.	2-163
q_vurgent	Posts a message at the head of a variable-length message queue.	2-165
rn_create	Creates a memory region.	2-167
rn_delete	Deletes a memory region.	2-170
rn_getseg	Allocates a memory segment to the calling task.	2-172
rn_ident	Obtains the region identifier of the named region.	2-174
rn_info	Queries a Region Object.	2-176
rn_retseg	Returns a memory segment to the region from which it was allocated.	2-178
sm_av	(pSOS+m kernel only) Asynchronously releases a semaphore token.	2-180
sm_create	Creates a semaphore.	2-182
sm_delete	Deletes a semaphore.	2-185
sm_ident	Obtains a semaphore identifier.	2-187
sm_info	Queries a Semaphore Object.	2-189
sm_notify	Registers the events for notification of semaphore availability.	2-191
sm_p	Acquires a semaphore token.	2-193

TABLE 2-1 pSOS+ System Calls (Continued)

Name	Description	Page
sm_v	Releases a semaphore token.	2-195
sys_info	Obtains pSOS+ system information.	2-197
t_addvar	Adds a new task variable to a task.	2-201
t_create	Creates a task.	2-203
t_delete	Deletes a task.	2-208
t_delvar	Deletes a task variable.	2-211
t_getreg	Gets a task's notepad register.	2-213
t_ident	Obtains the task identifier of the named task.	2-215
t_info	Queries a Task Object.	2-217
t_mode	Gets or changes the calling task's execution mode.	2-221
t_restart	Forces a task to start over regardless of its current state.	2-225
t_resume	Resumes a suspended task.	2-227
t_setpri	Gets and optionally changes a task's priority.	2-229
t_setreg	Sets a task's notepad register.	2-231
t_start	Starts a task.	2-233
t_suspend	Suspends a task until a t_resume call is made for the suspended task.	2-237
t_tslice	Gets and optionally changes its own or another task's timeslice quantum.	2-239
tm_cancel	Cancels an armed timer.	2-241
tm_evafter	Sends events to the calling task after a specified interval.	2-243
tm_evevery	Sends events to the calling task at periodic intervals.	2-245
tm_evwhen	Sends events to the calling task at the specified time.	2-247
tm_get	Obtains the system's current version of the date and time.	2-250
tm_getticks	Returns the elapsed tick count since the initialization of pSOS+.	2-252

TABLE 2-1 pSOS+ System Calls (Continued)

Name	Description	Page
tm_set	Sets or resets the system's version of the date and time.	2-254
tm_tick	Announces a clock tick to the pSOS+ kernel.	2-256
tm_wkafter	Blocks the calling task and wakes it after a specified interval.	2-258
tm_wkwhen	Blocks the calling task and wakes it at a specified time.	2-260
tsd_create	Creates a task-specific data (TSD) object.	2-262
tsd_delete	Deletes a task-specific data (TSD) Object	2-266
tsd_getval	Obtains a task's task-specific data (TSD) array entry value associated with the given TSD object.	2-268
tsd_ident	Obtains the index associated with a named task-specific data (TSD) object.	2-270
tsd_info	Queries a Task-Specific Data Entry Object.	2-272
tsd_setval	Sets and gets a task's data value for a Task-Specific Data (TSD) Object.	2-274

as_catch

Specifies an asynchronous signal routine (ASR).

```
#include <psos.h>
unsigned long as_catch(
   void (* start_addr) (),   /* ASR address */
   unsigned long mode        /* ASR attributes */
   )
```


Description

This system call allows a task to specify an asynchronous signal routine (ASR) to handle asynchronous signals. as_catch() supplies the starting address of the task's ASR, and its initial execution mode. If the input ASR address is zero, then the caller is deemed to have an invalid ASR, and any signals sent to it will be rejected.

A task's ASR gains control much like an ISR. If a task has pending signals (sent via as_send()), then the next time the task is dispatched to run, it will be forced to first execute the task's specified ASR. A task executes its ASR according to the mode supplied by the as_catch() call (for example, Non-preemptible, Time-slicing enabled, etc.) Upon entry to the ASR, all pending signals — including all those received since the last ASR invocation — are passed as an argument to the ASR. In addition, a stack frame is built to facilitate the return from the ASR.

as_catch() replaces any previous ASR for the calling task. Therefore, a task can have only one ASR at any time. An ASR must exit using the as_return() system call.

Arguments

start_addr	Specifies the address of the ASR.
mode	Specifies the ASR's attributes. mode is formed by OR-ing the following symbolic constants (one from each pair), which are defined in <psos.h>. For instance, to specify that the ASR should have preemption turned off, you place the symbolic constant T_NOPREEMPT in mode. To specify that the ASR should have preemption turned off and roundrobin by time-slicing turned on, you place both T_NOPREEMPT and T_TSLICE in mode, using the following syntax: T_NOPREEMPT | T_TSLICE

T_PREEMPT	ASR is preemptible.
T_NOPREEMPT	ASR is non-preemptible.
T_TSLICE	ASR can be time-sliced.
T_NOTSLICE	ASR cannot be time-sliced.
T_ASR	ASR nesting enabled.
T_NOASR	ASR nesting disabled. If T_ASR is set, then the ASR should be programmed to be re-entrant. If T_NOASR is set, the ASR is prevented from being re-entered as a result of another as_send() call made to that task.
T_USER	ASR runs in user mode.
T_SUPV	ASR runs in supervisor mode. See *User and Supervisor Modes* under *Target*.
T_ISR	Interrupts are enabled while ASR runs.
T_NOISR	Interrupts are disabled while ASR runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.
T_LEVELMASK0 through T_LEVELMASK*n*	Certain interrupts are disabled while ASR runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.

Target

User and Supervisor Modes

You use the symbolic constants T_USER and T_SUPV on each processor as follows:

68K CF ARM

On 68K, ColdFire, and ARM processors, you must place one of the symbolic constants T_USER or T_SUPV in mode to specify the ASR's processor mode.

PPC
960
MIPS
x86

On PowerPC, 960, MIPS and x86 processors, ASRs execute in supervisor mode only. Hence the symbolic constants T_USER and T_SUPV are ignored.

Interrupt Control

Interrupt control means that while an ASR is executing, hardware interrupts are disabled. On some processors, you can disable all interrupts at or below a certain interrupt level and enable all interrupts above that level. On other processors you can simply specify that all interrupts are either enabled or disabled. Details are provided below:

68K
CF
960

On 68K, ColdFire, and 960 processors, you can specify the hardware interrupt levels to disable. For instance, on 960 processors, where interrupts have 32 different levels, you could specify that all interrupts at level 15 or below are disabled. Note that interrupt levels have no relation to task priority levels.

Disabling certain interrupt levels is called masking. You set an ASR's interrupt mask level by placing any symbolic constant between T_LEVELMASK0 and T_LEVELMASK*n* in the mode argument, where *n* is the highest available interrupt level.

On 68K and ColdFire processors, the highest interrupt level is 7 and the lowest is 0. On 960 processors, the highest interrupt level is 31 and the lowest is 0.

PPC
ARM
x86
MIPS

On PowerPC, ARM, x86, and MIPS processors, you can simply enable or disable all hardware interrupts. You do this by placing either T_ISR or T_NOISR in the mode argument.

How Signals Are Passed to the ASR

The method by which signals are passed to the ASR is processor-specific:

68K — On 68K processors, signals are passed to the ASR in the D0.L register.

PPC ARM MIPS CF — On PowerPC, ARM, MIPS and ColdFire processors, signals are passed to the ASR as an argument, as if the ASR were a C function, following target-specific calling conventions.

960 — On 960 processors, signals are passed to the ASR in the G0 register.

x86 — On x86 processors, signals are passed to the ASR in the EAX register.

Return Value

This system call always returns 0.

Error Codes

None.

Notes

1. An invalid ASR (for example, `start_addr = 0`) should not be confused with the ASR attribute `T_NOASR`. If a task's ASR is invalid, then an `as_send()` call directed to it will be rejected and returned with an error; whereas, the `T_NOASR` attribute simply defers the ASR's execution, with any intervening signals sent to it left pending.
2. A normal task would call `as_catch()` only once, and usually as part of its initialization sequence. Before the first `as_catch()` call, a task is initialized by the pSOS+ kernel to have an invalid ASR.

Multiprocessor Considerations

None. The actions performed by `as_catch()` are entirely confined to the local node, although asynchronous signals can be sent from remote nodes.

Callable From

- Task

See Also

as_send, *as_return*, *as_notify*

as_notify

Registers the events for notification of signal.

```
#include <psos.h>
unsigned long as_notify(
    unsigned long events            /* bit-encoded events */
    )
```

Description

This system call registers a set of bit-encoded events for the calling task that are used to notify the posting of asynchronous signals to the task. The event notification mechanism provides a way for a task to synchronously wait for asynchronous signals via an `ev_receive()` call.

Arguments

`events`	Contains a list of bit-encoded events. A value of 0 turns off the event notification mechanism.

Return Value

This call always returns 0.

Error Codes

None.

Notes

1. When event notification for asynchronous signal has been requested by a task, the notification may be generated even if the task has not installed a handler for asynchronous signals.

Multiprocessor Considerations

None, `as_notify()` can be called only from the local node.

Callable From

- Task

See Also

as_catch, *as_send*, *q_notify*, *q_vnotify*, *sm_notify*, *ev_receive*

as_return

Returns from an asynchronous signal routine (ASR).

```
#include <psos.h>
unsigned long as_return();
```

Description

This system call must be used by a task's ASR to exit and return to the original flow of execution of the task. The purpose of this call is to enable the pSOS+ kernel to restore the task to its state before the ASR. `as_return()` cannot be called except from an ASR.

This call is analogous to the `i_return()` call, which enables an Interrupt Service Routine (ISR) to return to the interrupted flow of execution properly.

Target

Restoring CPU Registers

An ASR is responsible for restoring CPU registers to their previous state before exiting via `as_return()`. The exact way in which this happens varies from processor to processor. On most processors, the ASR is written in assembly language, so you the programmer must take care to restore the registers. On processors like PowerPC, MIPS, ColdFire and ARM, an ASR can be written in C, and the pSOS+ kernel restores the registers. Processor-specific information on restoring registers prior to `as_return()` is provided below:

On 68K processors, an ASR is responsible for saving and restoring all CPU registers it uses, including stack pointers. The one exception to this rule is the register D0.L, which is restored by the pSOS+ kernel. On 68K processors, an ASR can be written only in assembly language, and `as_return()` is an invalid system call. The skeleton for writing an ASR on 68K processors is exemplified below:

```
movem.l d1-d7/a0-a6, -(sp) ; save needed registers on stack
... BODY OF ASR ...
movem.l (sp)+, d1-d7/a0-a6 ; restore registers
move.l  #49,d0             ; load as_return function code
trap    #11                ; and make system call
```

PPC ARM MIPS CF

On PowerPC, ARM, MIPS and ColdFire processors, an ASR can be written in a high-level language such as C. The non-volatile registers will be saved by the pSOS+ kernel prior to an ASR and restored within the `as_return()` call.

960

On 960 processors, an ASR is responsible for saving and restoring all CPU registers it uses, including stack pointers. The one exception to this rule is the register g0, which is restored by the pSOS+ kernel. On 960 processors, an ASR can be written only in assembly language.

x86

On x86 processors, an ASR is responsible for saving and restoring all CPU registers it uses, including stack pointers. The one exception to this rule is the register EAX, which is restored by the pSOS+ kernel. On x86 processors, an ASR can be written only in assembly language.

Return Value

If successful, this system call never returns. An error code is generated on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. The actions performed by `as_return()` are confined entirely to the local node.

Callable From

- ASR

See Also

`as_catch`, *`as_send`*

as_send

Sends asynchronous signals to a task.

```
#include <psos.h>
unsigned long as_send(
   unsigned long tid,       /* target task ID */
   unsigned long signals    /* bit-encoded signal list */
   )
```

Description

This system call sends asynchronous signals to a task. Additionally, if signal notification via event has been requested, the notification events are also posted to the task. The purpose of these signals is to force a task to break from its normal flow of execution and execute its Asynchronous Signal Routine (ASR).

Asynchronous signals are like software interrupts, with ASRs taking on the role of ISRs. Unlike an interrupt, which is serviced almost immediately, an asynchronous signal does not immediately affect the state of the task. An `as_send()` call is serviced only when the task is next dispatched to run (and that depends on the state of the task and its priority).

Each task has 32 signals. These signals are encoded bit-wise in a single long word.

Like events, signals are neither queued nor counted. For example, if three identical signals are sent to a task before its ASR has a chance to execute, the three signals have the same effect as one.

Arguments

`tid`	Specifies the task to receive the signals.
`signals`	Contains the bit-encoded signals.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. When an ASR starts execution, all pending asynchronous signals (since its last invocation) are passed to it as an argument.
2. `as_send()` does not trigger the ASR handler if `signals` is 0. However, any notification events will be posted if they had been registered via a prior `as_notify()` call.
3. If the task is executing the task deletion callout code at the time the `as_send()` call is made, no signals or events are posted to the task.

Multiprocessor Considerations

If `tid` identifies a global task that resides on another processor node, the pSOS+ kernel internally makes a remote system call (RSC) to that remote node to send the asynchronous signal to the task.

Callable From

- Task.
- ISR, if the targeted task is local to the node from which the `as_send()` call is made.
- KI, if the targeted task is local to the node from which the `as_send()` call is made.
- Callout, if the targeted task is local to the node from which the `as_send()` call is made.

See Also

`as_catch`, *`as_notify`*, *`ev_send`*

co_register Registers a callout handler.

```
#include <psos.h>
unsigned long co_register(
   unsigned long type,             /* Type of callout */
   void (*func)(void *, CO_INFO *), /* Callout function */
   void *arg,                      /* Argument passed to callout */
   unsigned long *coid             /* Callout ID */
   )
```

Description

This system call registers a task startup, restart, or deletion callout handler. Any registered task startup and restart callouts are called in the order they were registered, just before pSOS+ passes control to the task's start address. Any registered task deletion callouts are called in the reverse order of their registration, just before the task is deleted.

The callout function always executes in the cotext of the task being started, restarted or deleted. It is invoked with two arguments: the first argument is the argument `arg` that was passed to `co_register`, and the second argument is pointer to `CO_INFO` structure defined in <psos.h>, that has two fields, as follows:

```
tid   ID of the task being started, restarted or deleted
ptid  ID of the task that performed t_start, t_restart or t_delete
```

Arguments

`type`	Specifies the type of callout being registered, and is one of the following symbolic constants defined in <psos.h>	
	`CO_START`	Task startup callout is being registered
	`CO_RESTART`	Task restart callout is being registered
	`CO_DELETE`	Task delete callout is being registered
`func`	Specifies the address of the callout function.	
`arg`	Specifies the argument that must be passed to the callout function.	
`coid`	Points to the variable where `co_register()` stores the ID of the registered callout.	

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The `arg` argument passed to `co_register()` is passed as it is to the callout function. It can be an integer value or a pointer. If `arg` is a pointer, the memory it is pointing to shall be allocated in an area that is accessible to the callout at all times while the callout is registered; pSOS+ does not make a copy of the memory area that is pointed to by `arg` at the time of `co_register()` call.
2. The task delete callout will not be executed when deleting a task that has been created but not started.
3. Any asynchronous signals posted to the task while a delete callout is in progress as a result of termination of that task, are ignored by pSOS+.
4. Any attempts to restart or delete a task while a delete callout is in progress as a result of termination of that task, result in an error.
5. It is possible to restart or delete a task while it is in the middle of executing a task startup or restart callout.
6. Callouts registered via `co_register()` service always execute in the context of a task and hence there is no restriction (with the exception of `co_unregister()` service) as to which pSOSystem services may or may not be invoked from a callout. However, care must be taken to keep the callout code concise since, once registered, a callout will be executed for all the tasks that subsequently get started, restarted, or deleted, as the case may be. Also, care must be taken to properly free up any resources allocated by a task deletion callout since any resources not freed therein may be lost forever.

Multiprocessor Considerations

Callouts can be registered only for the local node.

Callable From

- Task

See Also

co_unregister, *t_start*, *t_restart*, *t_delete*

co_unregister Un-registers a callout handler.

```
#include <psos.h>
unsigned long co_unregister(
    unsigned long coid,             /* Callout ID */
    unsigned long flags,            /* Flags */
    unsigned long timeout           /* Timeout value */
    )
```


Description

This system call un-registers a task startup, restart, or deletion callout handler, that had been registered earlier via `co_register()` service. The un-registration can occur only if there are no callouts of that type are in progress at the time of `co_unregister()` call. The calling task may choose to block until the callout un-registration is successful by specifying the `CO_WAIT` flag, and an optional timeout value. Alternatively, the calling task may choose not to wait, if the un-registration cannot be performed immediately, by specifying the `CO_NOWAIT` flag.

Arguments

`coid`	ID of the callout being un-registered.	
`flags`	Specifies whether co_unregister would wait for any in-progress call-outs of type similar to being un-registered to complete. It should have one of the following values (defined in <psos.h>):	
	`CO_WAIT`	Block until callout can be un-registered successfully.
	`CO_NOWAIT`	Return with error code if the callout cannot be un-registered immediately.
`timeout`	Specifies the timeout interval, in units of clock ticks. If `timeout` is 0, then co_unregister() will wait forever; `timeout` will be ignored if `CO_NOWAIT` flag is specified.	

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The `co_unregister()` service shall not be invoked from the body of the call-out function itself -- doing so will certainly result in a deadlock.

Multiprocessor Considerations

Callouts can be un-registered only for the local node.

Callable From

- Task

See Also

`co_register`, *`t_start`*, *`t_restart`*, *`t_delete`*

cv_abroadcast

Asynchronously awakens all the tasks waiting on a condition variable.

```
#include <psos.h>
unsigned long cv_abroadcast(
   unsigned long cvid,    /* condition variable identifier */
   )
```


Description

This system call asynchronously removes all tasks from a condition-variable's wait queue. It is identical to `cv_broadcast()` except that the call is made asynchronously. Refer to the description of `cv_broadcast()` for further information.

Arguments

`cvid`	Specifies the ID of the condition variable.

Return Value

When called in a system running the pSOS+m kernel, this call returns 0 on success or an error code on failure. The pSOS+ kernel (the single processor version) always returns `ERR_SSFN`.

Error Codes

Should the call fail, if present, the node's `MC_ASYNCERR` routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the `MC_ASYNCERR` routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is only supported by the pSOS+m kernel.
2. The calling task can be preempted as a result of this call.

Multiprocessor Considerations

1. If `cvid` identifies a global condition variable residing on another processor node, then the pSOS+ m kernel will internally make an RSC to that remote node to broadcast to the condition variable.

2. If the guarding mutex is not locked and the task thus awakened by this call does not reside on the same node as the condition variable, the kernel on the condition variable's node will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and give it the guarding mutex. Thus, a `cv_abroadcast()` call, whether it is to a local or remote condition variable, may cause pSOS+ activities on another node.

Callable From

- Task
- ISR, if the condition variable is local to the node from which the `cv_abroadcast()` call is made.
- KI, if the condition variable is local to the node from which the `cv_abroadcast()` call is made.
- Callout, if the condition variable is local to the node from which the `cv_abroadcast()` call is made.

See Also

`cv_broadcast`, *`cv_asignal`*, *`cv_wait`*

cv_asignal Asynchronously signals a condition variable.

```
#include <psos.h>
unsigned long cv_asignal(
   unsigned long cvid   /* condition variable ID */
   )
```


Description

This system call asynchronously removes the task at the head of a condition-variable's queue. It is identical to `cv_signal()` except that the call is made asynchronously. Refer to the description of `cv_signal()` for further information.

Arguments

`cvid`	Specifies the condition variable ID.

Return Value

When called in a system running the pSOS+m kernel, this call returns 0 on success or an error code on failure. The pSOS+ kernel (the single processor version) always returns `ERR_SSFN`.

Error Codes

Should the call fail, if present, the node's `MC_ASYNCERR` routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the `MC_ASYNCERR` routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+ m kernel.
2. The calling task can be preempted as a result of this call.

Multiprocessor Considerations

1. If `cvid` identifies a global condition variable residing on another processor node, then the pSOS+ m kernel will internally make an RSC to that remote node to signal the condition variable.

2. If the guarding mutex is not locked and the task thus awakened by this call does not reside on the same node as the condition variable, the kernel on the condition variable's node will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and give it the guarding mutex. Thus, a `cv_asignal()` call, whether it is to a local or remote condition variable, may cause pSOS+ activities on another node.

Callable From

- Task.
- ISR, if the condition variable is local to the node from which the `cv_asignal()` call is made.
- KI, if the condition variable is local to the node from which the `cv_asignal()` call is made.
- Callout, if the condition variable is local to the node from which the `cv_asignal()` call is made.

See Also

`cv_signal`, *`cv_abroadcast`*, *`cv_wait`*

cv_broadcast Awakens all the tasks waiting on a condition variable.

```
#include <psos.h>
unsigned long cv_broadcast(
   unsigned long cvid    /* condition variable identifier */
   )
```


Description

This system call removes all the tasks from a condition variable's wait queue. If the mutex guarding the condition variable is unlocked, the task at the head of the wait queue is given ownership of the guarding mutex and unblocked. The remaining tasks, including the task at the head of the wait queue if the guarding mutex is locked, are placed in the guarding mutex's wait queue.

If there are no tasks waiting, then `cv_broadcast()` does not do anything.

Arguments

`cvid`	Specifies the ID of the condition variable.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If called from a task, `cv_broadcast()` may cause the calling task to be preempted.
2. Tasks are removed one by one from the condition variable wait queue and added to the guarding mutex wait queue according to the queueing policy of the mutex. Thus, if the condition variable uses FIFO queueing while the mutex uses priority queueing, the order of the task relative to one another may change as they migrate from the condition variable wait queue to the mutex wait queue.

Multiprocessor Considerations

1. If `cvid` identifies a global condition variable residing on another processor node, then the pSOS+ m kernel will internally make an RSC to that remote node to broadcast to the condition variable.

2. If the guarding mutex is not locked, and the task thus awakened by this call does not reside on the same node as the condition variable, the kernel on the condition variable's node will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and, if appropriate, give it the guarding mutex. Thus, a `cv_broadcast()` call, whether it is to a local or remote condition variable, may cause pSOS+ activities on another node.

Callable From

- Task.
- ISR, if the condition variable is local to the node from which the `cv_broadcast()` call is made.
- KI, if the condition variable is local to the node from which the `cv_broadcast()` call is made.
- Callout, if the condition variable is local to the node from which the `cv_broadcast()` call is made.

See Also

`cv_wait`, *`cv_signal`*, *`mu_lock`*, *`cv_abroadcast`*

cv_create

Creates a condition variable.

```
#include <psos.h>
unsigned long cv_create(
   char name[4],          /* user assigned name */
   unsigned long flags,   /* condition variable attributes */
   unsigned long *cvid    /* condition variable identifier */
   )
```


Description

This system call creates a condition variable and initializes it according to the specifications supplied with the call.

Like all objects, a condition variable has a user-assigned name and a pSOS-assigned condition variable ID returned by `cv_create()`. Several flag bits specify the characteristics of the condition variable. Tasks can wait at the condition variable either by task priority or strictly FIFO.

Arguments

`name` Specifies the user-assigned name for the new condition variable.

`flags` Specifies the attributes of the condition variable. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`.

`CV_GLOBAL`	Condition variable is globally addressable by other nodes.
`CV_LOCAL`	Condition variable can be addressed only by the local node.
CV_PRIOR	The waiting tasks are queued in the order of their current priority.
CV_FIFO	The waiting tasks are queued in the `FIFO` order.

`cvid` Points to the variable where `cv_create()` stores the condition variable ID of the new condition variable.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a condition variable name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the condition variable name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate condition variable names. If duplicate names exist, `cv_ident()` can return the `cvid` of any condition variable with the duplicate name.
3. The maximum number of condition variables that can be simultaneously active is defined by the `kc_ncvar` entry in the pSOS+ Configuration Table.
4. A condition variable can be created only from the context of a task.

Multiprocessor Considerations

1. The `CV_GLOBAL` attribute is ignored by the single-processor version of the pSOS+ kernel.
2. The `CV_GLOBAL` attribute should be set only if the condition variable must be made known to other processor nodes in a multiprocessor configuration. If set, the condition variable's name and `cvid` are sent to the master node for entry in its Global Object Table.
3. If the `CV_GLOBAL` attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry `mc_nglbobj` then the condition variable is not created and `ERR_OBJTFULL` is returned.

Callable From

- Task

See Also

`cv_ident`, *`cv_delete`*

cv_delete Deletes a condition variable.

```
#include <psos.h>
unsigned long cv_delete (
   unsigned long cvid    /* condition variable identifier */
   )
```


Description

This system call deletes a condition variable specified by its ID (`cvid`) and frees the associated kernel resources. If there are tasks waiting at the condition variable, the tasks are unblocked and given an error code.

The calling task does not have to be the creator of the condition variable to be deleted. However, a condition variable must be deleted from the node on which it was created.

Arguments

`cvid`	Specifies the ID of the condition variable to delete.

Return Value

This system call returns 0 on success, and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A condition variable need not be deleted, even when it is no longer in use, except to allow reuse of the associated condition variable control block.
2. The calling task may be preempted, if a task waiting at the deleted condition variable has a higher priority.

Multiprocessor Considerations

1. If `cvid` identifies a global condition variable, `cv_delete()` will notify the master node so that the condition variable can be removed from its Global Object

Table. Thus, deletion of a global condition variable always causes activity on the master node.

Callable From

- Task

See Also

cv_create

cv_ident

Obtains the identifier of a named condition variable.

```
#include <psos.h>
unsigned long cv_ident(
   char name[4],          /* condition variable name */
   unsigned long node,    /* node number */
   unsigned long *cvid    /* condition variable identifier */
   )
```


Description

This system call enables the calling task to obtain the ID of a condition variable it only knows by name. This condition variable ID can be used in all other operations relating to the condition variable.

The pSOS+ kernel does not check for duplicate names. If multiple condition variables with the same name exist, either on the local node, or on a remote note in a multi-processor system, a `cv_ident()` call may return the ID of any one condition variable with the specified name. It can't be ascertained which ID is returned, though; to a certain extent it depends on the standard search order followed by pSOS+ kernel, as defined in the *pSOSystem System Concepts* manual.

Arguments

`name`	Specifies the user-assigned name of the condition variable.
`node`	Specifies the node selector. For multiprocessing systems, it is a search order specifier. In a single processor system, this argument must be 0.
`cvid`	Points to the variable where `cv_ident()` stores the ID of the named condition variable.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a condition variable name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the condition variable name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate condition variable names. If duplicate condition variable names exist, a `cv_ident()` call can return the `cvid` of any condition variable with the duplicate name.

Multiprocessor Considerations

1. `cv_ident()` converts a condition variable's name to its `cvid` by using a search order determined by the `node` input parameter, as described in *pSOSystem System Concepts*. Because condition variables created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, then the pSOS+m kernel makes a `cv_ident()` RSC to the master node.

Callable From

- Task

See Also

cv_create

cv_info

Queries about a Condition Variable Object.

```
#include <psos.h>
unsigned long cv_info(
   unsigned long cvid,              /* Condition Variable ID */
   struct cvinfo *buf,              /* Object Information buffer */
   )
```


Description

This system call returns object information for a given condition variable object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

cvid	Specifies the ID of the condition variable object being queried.
buf	Contains the Condition Variable information.

struct cvinfo includes the following members.

```
char              name[4];      /* Name of the Condition Variable */
unsigned long     flags;        /* Object's attributes */
unsigned long     wqlen;        /* No. of waiting tasks */
unsigned long     wtid;         /* ID of the first waiting task */
unsigned long     muid;         /* ID of the associated mutex */
```

name specifies the name of the condition variable. **wqlen** is the total number of tasks waiting on this object. **wtid** specifies the object ID of the first waiting task. **wtid** is zero if there are no waiting tasks. The object ID of the associated mutex is returned in **muid**. **muid** is meaning less when **wqlen** is 0.

flags has the attributes with which this object was created.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The value of the system configuration parameter, SC_PSOS_QUERY, should be set to YES in the file <sys_conf.h>, if the cv_info() call is to be used by the application.

Multiprocessor Considerations

cv_info() can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

cv_create

cv_signal Signals a condition variable.

```
#include <psos.h>
unsigned long cv_signal(
   unsigned long cvid    /* condition variable identifier */
   )
```


Description

This system call removes the task at the head of a condition variable's wait queue from that queue. If the mutex guarding the condition variable is unlocked, the task is given ownership of the guarding mutex and unblocked. If the mutex is already locked, then the task is placed in the guarding mutex's wait queue according to the queueing policy of the mutex.

If there are no tasks waiting at the condition variable, this call does nothing.

Arguments

`cvid`	Specifies the condition variable identifier.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If called from a task, `cv_signal()` may cause the calling task to be preempted.

Multiprocessor Considerations

1. If `cvid` identifies a global condition variable residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to signal the condition variable.
2. If the guarding mutex is not locked and the task thus awakened by this call does not reside on the same node as the condition variable, the kernel on the

condition variable's node will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and give it the guarding mutex. Thus, a `cv_signal()` call, whether it is to a local or remote condition variable, may cause pSOS+ activities on another node.

Callable From

- Task
- ISR, if the condition variable is local to the node from which the `cv_signal()` call is made.
- KI, if the condition variable is local to the node from which the `cv_signal()` call is made.
- Callout, if the condition variable is local to the node from which the `cv_signal()` call is made.

See Also

cv_broadcast, *cv_wait*, *mu_lock*, *mu_unlock*

cv_wait

Waits on a condition variable.

```
#include <psos.h>
unsigned long cv_wait(
   unsigned long cvid,     /* condition variable identifier */
   unsigned long muid,     /* mutex identifier */
   unsigned long timeout   /* timeout interval */
   )
```


Description

This system call is used by a task to wait on a condition variable. The guarding mutex must be owned by the caller. If tasks are waiting at the condition variable, then `muid` must specify the mutex already guarding the condition variable. Otherwise, any mutex may be specified.

`cv_wait()` unlocks the guarding mutex and blocks the calling task. Unless a timeout occurs, the calling task waits at the condition variable until removed from the wait queue by a `cv_signal()` or a `cv_broadcast()` call. When removed, ownership of the guarding mutex is given to the task if the guarding mutex is currently unlocked. Otherwise, the task waits for ownership of the guarding mutex according to the queueing policy of the mutex.

Arguments

`cvid`	Specifies the condition variable identifier.
`muid`	Specifies the guarding mutex identifier.
`timeout`	Specifies the maximum time the calling task will wait for release from the condition variable and subsequent reacquisition of the guarding mutex. If `timeout` is zero, then the calling task waits forever.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If an error occurs, the guarding mutex is always unlocked upon return from `cv_wait()` regardless of the type of error.
2. If a task transitions from the "waiting on condition variable" state to the "waiting to lock mutex" state, the timeout specified with the call carries over. For example, if the `cv_wait()` call is made with a timeout value of 10 ticks and the task spends 4 ticks waiting on condition variable before going to wait on mutex, the starting timeout value for the mutex wait will be 6 ticks.
3. If the condition variable was created with the `CV_FIFO` attribute, then the calling task is simply entered at the tail of the wait queue. If the condition variable was created with the `CV_PRIOR` attribute, then the calling task is inserted into the wait queue by priority.
4. Since `cv_wait()` implicitly unlocks and locks the given mutex, a call to `cv_wait()` may change the priority of the calling task or some other task in the system, depending on the mutex attributes, as specified in the description of the `mu_create()` system call.

Multiprocessor Considerations

1. The condition variable and its guarding mutex must reside on the same processor node.
2. If `cvid` identifies a global condition variable residing on another processor node, the local kernel will internally make a RSC to that remote node to wait at the condition variable. The pSOS+ kernel on the target node must use an agent to wait at the condition variable. If that node is temporarily out of agents, the call will fail. The number of agents on each node is defined by the `mc_agent` entry in the node's Multiprocessor Configuration Table.

Callable From

- Task

See Also

`cv_signal`, *`cv_broadcast`*

de_close

Closes an I/O device.

```
#include <psos.h>
unsigned long de_close(
   unsigned long dev,    /* major/minor device number */
   void *iopb,           /* I/O parameter block address */
   void *retval          /* return value */
   )
```


Description

The de_close() call invokes the device close routine of a pSOS+ device driver specified by the dev argument. The functionality of the device close routine is device-specific. For example, an RS-232 device driver close routine may signal a modem to hang up to signify the end of the connection.

The de_close() call, when used in conjunction with de_open(), can also be used to implement mutual exclusion. In this case, de_close() can be used to signal the end of a critical region for the device operation.

Arguments

dev	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Points to an I/O parameter block, the contents of which are driver-specific.
retval	Points to a variable that receives a driver-specific value returned by the driver.

Return Value

This call returns 0 on success, or an error code on failure. Besides the error codes listed below, other driver-specific errors may be returned.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_open

de_cntrl

Requests a special I/O device service.

```
#include <psos.h>
unsigned long de_cntrl(
   unsigned long dev,    /* major/minor device number */
   void *iopb,           /* I/O parameter block address */
   void *retval          /* return value */
   )
```


Description

The de_cntrl() call invokes the device control routine of a pSOS+ device driver specified by the dev argument. The functionality of a device control routine depends entirely on the device driver implementation. It can include anything that cannot be categorized under the other five I/O services. de_cntrl() for a device can be used to perform multiple input and output subfunctions. In such cases, extra parameters in the I/O parameter block can designate the subfunction.

Arguments

dev	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Points to an I/O parameter block, the contents of which are driver-specific.
retval	Points to a variable that receives a driver-specific value returned by the driver.

Return Code

This call returns 0 on success or an error code on failure. Beside the error codes listed below, other driver-specific errors may be returned.

Error Codes

Refer to Appendix A.

Notes

Examples of functions that are often performed by `de_cntrl()` include the following:

- For tty drivers, functions such as changing the baud rate and line-edit characters, enabling/disabling of typed character echo, and so on
- Reading device status information

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_read, *de_write*, *de_open*, *de_close*

de_init

Initializes an I/O device and its driver.

```
#include <psos.h>
unsigned long de_init(
   unsigned long dev,    /* major/minor device number */
   void *iopb,           /* I/O parameter block */
   void *retval,         /* return value */
   void **data_area      /* device data area */
   )
```


Description

The de_init() call invokes the device initialization routine of the pSOS+ device driver specified by the dev argument.

The drive init routine can perform one-time device initialization functions such as:

- Resetting the devices
- Setting the necessary programmable registers
- Allocating and/or initializing the driver's data area (for pointers, counters, and so on)
- Creating the messages queues, semaphores, and so on, that are needed for communication and synchronization
- Installing the interrupt vectors, if necessary

Arguments

dev	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Points to an I/O parameter block, the contents of which are driver-specific.
retval	Points to a variable that receives a driver-specific value returned by the driver.
data_area	This argument is no longer used, but it remains to support compatibility with older drivers and/or pSOS+ application code. The pSOS+ bindings store a value into the variable pointed to by data_area. Therefore, a dummy variable still must be allocated to prevent memory corruption.

Return Value

This call returns 0 on success, or an error code on failure. Besides the error codes listed below, other driver-specific errors may be returned.

Error Codes

Refer to Appendix A.

Notes

1. The pSOS+ kernel will automatically call `de_init()` during system initialization if device auto-initialization is enabled. Refer to the *pSOSystem System Concepts* manual for further details.
2. Normally `de_init()` is called once from the ROOT task for each configured device driver. This call is normally made before other driver services are used.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_open

de_open

Opens an I/O device.

```
#include <psos.h>
unsigned long de_open(
   unsigned long dev,   /* major/minor device number */
   void *iopb,          /* I/O parameter block address */
   void *retval         /* return value */
   )
```


Description

The de_open() call invokes the device open routine of a pSOS+ device driver specified by the dev argument.

The device open routine can be used to perform functions that need to be done before the I/O operations can be performed on the device. For example, an asynchronous serial device driver can reset communication parameters (such as baud rate and parity) to a known state for the channel being opened.

A device driver can also assign specific duties to the open routine that are not directly related to data transfer or device operations. For example, a device driver can use de_open() to enforce exclusive use of the device during several read and/or write operations.

Arguments

dev	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Points to an I/O parameter block, the contents of which are driver-specific.
retval	Points to a variable that receives a driver-specific value returned by the driver.

Return Value

This call returns 0 on success, or an error code on failure. Besides the error codes listed below, other driver-specific errors may be returned.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_close

de_read

Reads from an I/O device.

```
#include <psos.h>
unsigned long de_read(
   unsigned long dev,    /* major/minor device number */
   void *iopb,           /* I/O parameter block address */
   void *retval          /* return value */
   )
```


Description

The de_read() call is used to read data from a device. It invokes the device *read* routine of a pSOS+ device driver specified by the dev argument. This service normally requires additional parameters contained in the I/O parameter block, such as the address of a data area to hold the data and the number of data units to read.

Arguments

dev	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
iopb	Points to an I/O parameter block, the contents of which are driver-specific.
retval	Points to a variable that receives a driver-specific value returned by the driver. For example, it can hold the actual number of data units read.

Return Value

This call returns 0 on success, or an error code on failure. In addition to the error codes listed below, other driver-specific errors may be returned.

Error Codes

Refer to Appendix A.

Notes

For many interrupt-driven devices, de_read() starts an I/O transaction and blocks the calling task. Most of the I/O transaction can actually be performed in the

device's ISR. Upon completion of the transaction, the ISR unblocks the blocked task.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_write

de_write

Writes to an I/O device.

```
#include <psos.h>
unsigned long de_write(
   unsigned long dev,    /* major/minor device number */
   void *iopb,           /* I/O parameter block address */
   void *retval          /* return value */
   )
```


Description

The `de_write()` call is used to write to a device. It invokes the device write routine of a pSOS+ device driver specified by the `dev` argument. This service normally requires the additional parameters contained in the I/O parameter block, such as the address of the user's output data and the number of data units to write.

Arguments

`dev`	Specifies the major and minor device numbers, which are stored in the upper and lower 16 bits, respectively.
`iopb`	Points to an I/O parameter block, the contents of which are driver-specific.
`retval`	Points to a variable that receives a driver-specific value returned by the driver (the actual number of data units written, for example.)

Return Value

This call returns 0 on success, or an error code on failure. Besides the error codes listed below, other driver-specific errors can be returned.

Error Codes

Refer to Appendix A.

Notes

For many interrupt-driven devices, `de_write()` starts an I/O transaction and blocks the calling task. Most of the I/O transactions can actually be performed in

the device's ISR. Upon completion of the transaction, the ISR unblocks the blocked task.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

de_read

dnt_add

Adds a new device name to the DNT.

```
#include <psos.h>
unsigned long dnt_add(
   char *name,             /* device name */
   unsigned long devnum    /* major/minor device number */
   )
```


Description

The `dnt_add()` system call is used to add a new device name to the Device Name Table (DNT).

Arguments

name	Pointer to the null terminated device name.
devnum	Specifies the major/minor number associated with the device name. It need not correspond to an existing driver, i.e. the major number may correspond to an I/O Jump Table entry which is currently unbound.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Multiprocessor Considerations

None. An application may only modify the device name table on the local node.

Callable From

- Task

See Also

dnt_remove, *dnt_find*

dnt_find

Returns the major/minor number associated with a device name.

```
#include <psos.h>
unsigned long dnt_find(
   char *name,             /* device name */
   unsigned long *devnum   /* device number */
   )
```


Description

This system call searches the Device Name Table (DNT) for an entry matching name and, if found, stores the corresponding major/minor number into the variable pointed to by devnum.

Arguments

name	Pointer to the null terminated device name.
devnum	Points to the variable where dnt_find() stores the major/minor number associated with the device name.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only search the device name table on the local node.

Callable From

- Task

See Also

dnt_add, *dnt_remove*

dnt_remove

Removes a device name from the DNT.

```
#include <psos.h>
unsigned long dnt_remove(
   char *name   /* device name to remove */
   )
```


Description

The `dnt_remove()` system call searches the Device Name Table (DNT) for an entry matching `name`, and if found, removes that entry.

Arguments

`name`	Pointer to the null terminated device name to remove.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only modify the device name table on the local node.

Callable From

- Task

See Also

dnt_add, *dnt_find*

errno_addr

Obtains the address of the calling task's internal errno variable.

```
#include <psos.h>
unsigned long *errno_addr();
```

Description

This system call returns the address of the calling task's internal errno variable.

The pSOS+ kernel maintains an internal errno variable for every task. Whenever an error is detected by any pSOSystem component, the associated error code is stored into the running task's internal errno variable. The error code can then be retrieved by referencing the errno macro defined in the header file <psos.h> as follows:

```
#define errno (*(errno_addr())
```

For example, the following statement expands to include a call to errno_addr():

```
if (errno == ERR_NOMGB)
```

Return Value

This system call returns the address of the errno variable of the calling task.

Error Codes

None.

Notes

1. errno_addr() provides a unique errno value for each task while maintaining compatibility with industry standard library semantics. It should never be necessary to call errno_addr() directly from application code.
2. All pSOSystem components set a task's internal errno variable. However, for the pSOS+ kernel and pHILE+ file system manager, which return error values via the function return value, use of the errno macro is superfluous.
3. A successful system call does not clear the previous errno value. errno always contains the error code from the last unsuccessful call.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

Not Applicable.

ev_asend (pSOS+m kernel only) Asynchronously sends events to a task.

```
#include <psos.h>
unsigned long ev_asend(
   unsigned long tid,      /* target task identifier */
   unsigned long events    /* bit-encoded events */
   )
```

Description

This system call asynchronously sends events to a task. It is identical to ev_send() except the call is made asynchronously. Refer to the description of ev_send() for further information.

Arguments

tid	Specifies the task ID of the target task.
events	Contains a list of bit-encoded events.

Return Value

When called in a system running the pSOS+m kernel, this call always returns 0. The pSOS+ kernel (the single processor version) returns ERR_SSFN.

Error Codes

Should the call fail, if present, the node's MC_ASYNCERR routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the MC_ASYNCERR routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+m kernel.
2. The events sent to a non-waiting task, or those that do not match the events being waited for, are always left pending.
3. If the tid input argument identifies a task residing on the local processor node, the calling task may be preempted as a result of this call.

4. In a multiple-event wait situation, the `ev_send()` and `ev_receive()` pair of calls depend greatly on the temporal course of events. See Note 2 under `ev_receive()` for an example.

5. The pSOS+m kernel does not prevent the use of bits reserved for system use. However, for future compatibility, these bits should not be used.

Multiprocessor Considerations

If the `tid` input argument identifies a global task residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to send the specified `events` to that task.

Callable From

- Task

See Also

ev_send, *ev_receive*

ev_receive

Enables a task to wait for an event condition.

```
#include <psos.h>
unsigned long ev_receive(
   unsigned long events,    /* bit-encoded events */
   unsigned long flags,     /* event processing attributes */
   unsigned long timeout,   /* timeout delay */
   unsigned long *events_r  /* events received */
   )
```

Description

This service call enables a task to wait for an event condition. The event condition is a set of user-defined events and an ANY/ALL waiting condition qualifier. Each task can wait on 32 events, which are bit-encoded in a long word. An ALL condition occurs when all of the specified events are received. An ANY condition occurs when one or more of the specified events is received.

If the selected event condition is satisfied by events already pending, `ev_receive()` clears those events and returns. Otherwise, `ev_receive()` can return immediately with an error, wait until the requisite events have been received, or wait until a timeout occurs, depending on the `flags` argument.

If successful, `ev_receive()` returns the actual events captured by the call in the location pointed to by `events_r`.

Arguments

`events` Specifies the set of events. An `events` argument equal to 0 is a special case, where `ev_receive()` returns the pending events but leaves them pending. In this case, the other parameters are ignored.

`flags` Specifies the event processing attributes. `flags` is formed by ORing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that `ev_receive()` blocks until all events are satisfied, you place `EV_WAIT` and `EV_ALL` in `flags`, using the following syntax:

```
EV_WAIT | EV_ALL
```

To specify that `ev_receive()` blocks until at least one event is satisfied, you place `EV_WAIT` and `EV_ANY` in `flags`.

	EV_NOWAIT EV_WAIT	Return if the event condition is unsatisfied. Block until the event condition is satisfied. Selecting EV_NOWAIT is a convenient way to reset all or selected pending events. For example, an ev_receive() for events 1 and 2 unconditionally resets events 1 and 2.
	EV_ANY EV_ALL	Wait for ANY of the desired events. Wait for ALL of the desired events. A successful return with EV_ANY signifies that at least one specified event was captured. A successful return with the EV_ALL attribute signifies that all specified events have been captured.
timeout		If EV_WAIT is set, the timeout parameter specifies the timeout in units of clock ticks. If the value of timeout is 0, ev_receive() waits indefinitely. If EV_NOWAIT is set, the timeout argument is ignored.
events_r		Points to the variable where ev_receive() stores the actual events captured.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Events are not accumulated. No matter how many *identical* events are sent to the calling task before it calls ev_receive() for receiving the event, the result is the same as if one event were pending.
2. The ev_receive() call captures only the events that the caller selects. It captures each selected event once. If a pending event does not match a selected event, the pending event remains pending. Also, if a pending event was sent after an earlier event was used to match a selected event, the pending event remains pending. Consider the following example sequence:
   a. Task P has pending events 1 and 2.

b. With EV_ALL set, P calls ev_receive() for events 1, 3, and 8. Pending event 1 is cleared.

c. Task A sends events 1 and 8 to P.

d. Event 1 is made pending. Event 8 is used to match the wanted event.

e. Task B sends events 2, 3, and 5 to P. Event 2 has no effect because event 2 is already pending. Event 5 is unwanted and made pending. Event 3 is used to match a wanted event. The event condition is met, so P becomes ready to run.

f. Events 1, 2, and 5 are left pending.

g. Events 1, 3, and 8 are returned in events_r.

Multiprocessor Considerations

None. The actions performed by ev_receive() take place only on the local node (whether or not events come from other nodes).

Callable From

- Task

See Also

ev_send

ev_send

Sends events to a task.

```
#include <psos.h>
unsigned long ev_send(
   unsigned long tid,      /* target task identifier */
   unsigned long events    /* bit-encoded events */
   )
```


Description

This system call sends events to a task. If the target task is not waiting for events, the newly sent events are simply made pending. If the task is waiting for events, and the wait condition is fully satisfied as a result of the new events, then the task is unblocked and readied for execution. Otherwise, the task continues to wait. In either case, any of the events sent that do not match those waited on are always left pending.

Events are neither queued nor counted. For example, if three identical events are sent to a task before it issues a wait for that event, the three events have the same effect as one event.

Arguments

`tid`	Specifies the task identifier of the target task. When the tid is 0, events are sent to the running task on the local node.
`events`	Contains a list of bit-encoded events.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The events sent to a non-waiting task, or those that do not match the events being waited for, are always simply left pending.
2. If the caller is a task, it may be preempted as a result of this call.
3. In a multiple-event wait situation, the `ev_send()` and `ev_receive()` pair of calls are highly dependent on the temporal course of events. See Note 2 under `ev_receive()` for an example.
4. The pSOS+ kernel does not prevent the use of bits reserved for system use. However, for future compatibility, these bits should not be used.

Multiprocessor Considerations

If the `tid` input argument identifies a global task residing on another processor node, then the pSOS+ kernel internally makes an RSC to that remote node to send the input events to that task.

Callable From

- Task.
- ISR, if the targeted task is local to the node from which the `ev_send()` call is made.
- KI, if the targeted task is local to the node from which the `ev_send()` call is made.
- Callout, if the targeted task is local to the node from which the `ev_send()` call is made.

See Also

`ev_receive`

ioj_bind

Binds a particular I/O Jump Table entry and a driver.

```
#include <psos.h>
unsigned long ioj_bind (
   unsigned long major,   /* entry to bind */
   struct iojent *iojent  /* driver description */
   )
```


Description

This system call copies the structure pointed to by iojent into the I/O Jump Table entry specified by `major`. The `major` variable must be in the legal range and specify an unbound entry.

If the `IO_AUTOINIT` bit is set in the `flags` field within `iojent`, `ioj_bind()` calls the driver's `de_init()` function.

Arguments

`major`	Specifies the major number of the I/O Jump Table entry to bind.
`iojent`	Points to a structure which describes the device driver.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

If the driver's `de_init()` routine is called and returns an error, that error is passed back to the caller.

Notes

None.

Multiprocessor Considerations

None. An application may only modify the I/O Jump Table on the local node.

Callable From

- Task

See Also

ioj_bindany, *ioj_unbind*

ioj_bindany

Binds any I/O Jump Table entry to a driver.

```
#include <psos.h>
unsigned long ioj_bindany (
   struct iojent *iojent, /* driver description */
   unsigned long *major   /* major number of entry */
   )
```


Description

This system call locates an unbound I/O Jump Table entry and copies the structure pointed to by `iojent` into that entry. The major number of the entry is stored into the variable pointed to by `major`.

If the `IO_AUTOINIT` bit is set in the `flags` field within `iojent`, `ioj_bindany()` calls the driver's `de_init()` function.

Arguments

`iojent`	Points to a structure which describes the device driver.
`major`	Points to the variable where `ioj_bindany()` stores the major number of the I/O Jump Table entry that has been bound.

Return Value

This system returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

If the driver's `de_init()` routine is called and returns an error, that error is passed back to the caller.

Notes

None.

Multiprocessor Considerations

None. An application may only modify the I/O Jump Table on the local node.

Callable From

- Task

See Also

ioj_bind, *ioj_unbind*

ioj_getent

Obtain information about a particular I/O Jump Table entry.

```
#include <psos.h>
unsigned long ioj_getent (
   unsigned long major,   /* major no. of entry requested*/
   struct iojent *iojent  /* pointer to returned entry */
   )
```


Description

This system call copies the I/O Jump Table entry specified by `major` into the structure pointed to by `iojent`. The `major` variable must be in the legal range and specify a bound entry.

Arguments

`major`	Specifies the major number of the I/O Jump Table entry to obtain.
`iojent`	Points to a structure where the I/O Jump Table entry is returned.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only access the I/O Jump Table on the local node.

Callable From

- Task

See Also

ioj_bind

ioj_lock Increments the lock count of an I/O Jump Table entry.

```
#include <psos.h>
unsigned long ioj_lock (
   unsigned long major   /* major number of entry */
   )
```


Description

This system call increments the lock count of an I/O Jump Table entry. Locking an entry prevents any inadvertent unbinding of an I/O Jump Table entry.

Arguments

major	Specifies the major number of the I/O Jump Table entry.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only lock an I/O Jump Table entry on the local node.

Callable From

- Task

See Also

ioj_unlock

ioj_unbind

Unbinds an I/O Jump Table entry and a driver.

```
#include <psos.h>
unsigned long ioj_unbind (
   unsigned long major,   /* major number of entry */
   )
```

Description

This system call unbinds an I/O Jump Table entry and a driver. Once an entry has been unbound, all subsequent references to that major number return the error code ERR_NBOUND.

If major specifies an illegal or unbound entry, or if the entry is protected, an error is returned. Otherwise, the entry is unbound. An entry is protected when its lock count is not zero.

Arguments

major	Specifies the major number of the entry to unbind.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Unbinding an entry has no effect on any other pSOSystem data structures such as open connections, mounted volumes, the content of the device name table, open streams, etc.
2. pSOS+ does not verify if there are any open connections to or mounted volumes on the device being unbound. Any open connections to the device shall be closed, and mounted volumes be unmounted prior to unbinding the device.

Multiprocessor Considerations

None. An application may only modify the I/O Jump Table on the local node.

Callable From

- Task

See Also

ioj_bind, *ioj_bindany*

ioj_unlock Decrements the lock count of an I/O Jump Table entry.

```
#include <psos.h>
unsigned long ioj_unlock (
   unsigned long major   /* major number of entry */
   )
```

Description

This system call decrements the lock count of an I/O Jump Table entry. Unlocking an entry permits unbinding of an I/O Jump Table entry.

Arguments

major	Specifies the major number of the I/O Jump Table entry.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only unlock an I/O Jump Table entry on the local node.

Callable From

- Task

See Also

ioj_lock

k_fatal

Aborts and enters fatal error handling mode.

```
#include <psos.h>
void k_fatal(
   unsigned long err_code,   /* user's error code */
   unsigned long flags       /* fatal condition attributes */
   )
```


Description

This system call allows the user application to pass control to the user-defined fatal error handler in the event of a nonrecoverable failure. `k_fatal()` forces a nonrecoverable shutdown of the pSOS+ environment and never returns to the caller.

Arguments

`err_code` Specifies a user-defined failure code that is passed to the fatal error handler. The failure code must be at least 0x20000000.

`flags` The `flags` argument is ignored in the single-processor version of the pSOS+ kernel. In a multiprocessor system, the `flags` argument is used to determine whether the local node should be shut down or a system-wide shutdown should occur. `flags` is formed by selecting one of the following symbolic constants, which are defined in `<psos.h>` (see *Multiprocessor Considerations*).

- `K_GLOBAL` `k_fatal()` invocation causes global system shutdown.
- `K_LOCAL` `k_fatal()` invocation causes local node to shut down.

If the value of `flags` is `K_GLOBAL`, a global shutdown packet is sent to the master, which then sends a shutdown packet to every other node in the system.

Return Value

This call never returns to the caller.

Notes

1. The shutdown procedure is a procedure whereby pSOS+ attempts to halt execution in the most orderly manner possible. The pSOS+ kernel first examines the pSOS+ Configuration Table entry `kc_fatal`. If this entry is nonzero, the pSOS+ kernel jumps to this address. If `kc_fatal` is zero, and the pROBE+ System Debug/Analyzer is present, then the pSOS+ kernel passes control to the System Failure entry of the pROBE+ debugger. For a description of the pROBE+ debugger behavior in this case, refer to the *pROBE+ User's Manual*. Finally, if the pROBE+ debugger is absent, the pSOS+ kernel internally executes an illegal instruction to cause a deliberate illegal instruction exception. This passes control to a ROM monitor or other low-level debug tool.
2. `k_fatal()` is not the only mechanism by which control is passed to the fatal error handler. It can also receive control following an internal pSOS+ fatal error or, in multiprocessor systems, a shutdown packet from the master node.

Multiprocessor Considerations

In a multiprocessor system, `k_fatal()` can be used to implement a system-wide abort or shutdown. In this case, `K_GLOBAL` should be set. This causes a global shutdown packet to go to the master node, which sends a shutdown packet to every node in the system.

Callable From

- Task
- KI
- Callout
- ISR

See Also

k_terminate

k_terminate

Terminates a node other than the master node.

```
#include <psos.h>
unsigned long k_terminate (
   unsigned long node,   /* node to terminate */
   unsigned long fcode,  /* failure code */
   unsigned long flags   /* unused */
   )
```


Description

This system call enables the user application to shut down a node that it believes has failed or is operating incorrectly. `k_terminate()` causes the specified node to receive a shutdown packet and all other nodes to receive notification of the specified node's failure.

Arguments

`node`	Specifies the node number of the node to shut down. It cannot be the master node.
`fcode`	Specifies a user-defined failure code. It must be at least 0x20000000.
`flags`	Unused.

Return Value

This system returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `k_terminate()` can be used to terminate the node from which it is called. In most cases the results are the same as a `k_fatal()` call. However, it is implemented differently. Whereas `k_fatal()` immediately enters the fatal error handler, `k_terminate()` causes a packet to be sent to the master node, which then sends a shutdown packet to the calling node. If the calling node cannot

communicate with the master, then the KI presumably calls `k_fatal()` anyway. It is preferable to use `k_fatal()` when the failed node is known to be the local node.

2. A `k_fatal()` call made with the `K_GLOBAL` flag set should be used to shut down the entire system including the master node.

Multiprocessor Considerations

`k_terminate()` is primarily intended for use in multiprocessor systems although it can be used to terminate execution in a simple processor system.

Callable From

- Task
- ISR
- KI
- Callout

See Also

`k_fatal`

m_ext2int

Converts an external address into an internal address.

```
#include <psos.h>
unsigned long m_ext2int(
   void *ext_addr,   /* external reference */
   void **int_addr   /* local reference */
   )
```


Description

This system call converts an external address into an internal address corresponding to the calling node. A typical use for this conversion is by a node that has received an address from another node that resides in a dual-ported memory zone.

`m_ext2int()` is relevant only to systems with multiple processors connected by dual-ported memory on a memory bus. Other users can disregard this call.

Arguments

`ext_addr`	Specifies the external address.
`int_addr`	Points to the variable where `m_ext2int()` stores the resultant internal address. If the external address is within a dual-ported zone whose p-port is tied to the calling node, then the internal address will be different. In all other cases, the internal and external addresses will be the same.

Return Value

This system call always returns 0.

Error Codes

None.

Notes

1. For descriptions of internal and external addresses and dual-ported memory considerations, see the *pSOSystem System Concepts* manual.
2. Be careful about structures that straddle the boundary of a dual-port zone, because the address range for the structure may contain a discontinuity.

Multiprocessor Considerations

None. Although `m_ext2int()` is primarily used in multiprocessor systems, its action is restricted to the local node.

Callable From

- Task
- ISR
- KI
- Callout

See Also

m_int2ext

m_int2ext

Converts an internal address into an external address.

```
#include <psos.h>
unsigned long m_int2ext(
   void *int_addr,   /* local reference */
   void **ext_addr   /* external reference */
   )
```


Description

When a node on a multiprocessor system passes an address that resides within a dual-ported zone, it first must convert the address by calling `m_int2ext()`. This call applies to systems with multiple processors that are connected by dual-ported memory on a memory bus.

Arguments

`int_addr`	Specifies the internal address.
`ext_addr`	Points to the variable where `m_int2ext()` stores the resultant external address. If the internal address is within a dual-ported zone whose p-port is tied to the calling node, the external address is different. In all other cases, the internal and external addresses are the same.

Return Value

This call always returns 0.

Error Codes

None.

Notes

Be careful about structures that straddle the boundary of a dual-port zone, because the structure's address range could contain a discontinuity.

Multiprocessor Considerations

None. Although used in multiprocessor systems, `m_int2ext()` executes on the local node.

Callable From

- Task
- ISR
- KI
- Callout

See Also

`m_ext2int`

mu_create Creates a mutex.

```
#include <psos.h>
unsigned long mu_create(
   char name[4],         /* user assigned name */
   unsigned long flags,  /* mutex attributes */
   unsigned long ceiling,/* ceiling priority */
   unsigned long *muid   /* newly created mutex id */
   )
```


Description

This system call creates a mutex and initializes it according to the specifications supplied with the call.

Arguments

`name` Specifies the user-assigned name for the new mutex.

`flags` Specifies the mutex attributes. `flags` is formed by OR-ing the following symbolic constants (one from each set), which are defined in `<psos.h>`. For instance, to specify that a mutex is globally addressable, you place the symbolic constant `MU_GLOBAL` in `flags`. To specify that the mutex is globally addressable and that the waiting tasks are to be queued in the FIFO order, you place both `MU_GLOBAL` and `MU_FIFO` in `flags`, using the following syntax:

```
MU_GLOBAL| MU_FIFO
```

`MU_GLOBAL`	Mutex is globally addressable by other nodes of a multiprocessor system. The single-processor version of the pSOS+ kernel ignores `MU_GLOBAL`.
`MU_LOCAL`	Mutex can be addressed only by the local node where it was created.
`MU_RECURSIVE`	Mutex can be acquired in a recursive fashion (see *mu_lock* *on page 2-96*).
`MU_NORECURSIVE`	Mutex cannot be acquired in a recursive fashion (see *mu_lock* *on page 2-96*).

	MU_FIFO	The waiting tasks are queued in the FIFO order. This flag can be specified with MU_PRIO_NONE flag, but not with MU_PRIO_INHERIT or MU_PRIO_PROTECT.
	MU_PRIOR	The waiting tasks are queued in decreasing order of their current priority.
	MU_PRIO_NONE	Mutex does not protect against unbounded priority inversion.
	MU_PRIO_INHERIT	Mutex uses the priority inheritance protocol to prevent unbounded priority inversion.
	MU_PRIO_PROTECT	Mutex uses the priority protect protocol to prevent unbounded priority inversion.
ceiling	Specifies the ceiling priority of the mutex and is used only when the MU_PRIO_PROTECT flag has been specified.	
muid	Points to the variable which contains the ID of the newly created mutex on successful creation of the mutex object.	

Return Value

This call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The maximum number of mutexes that can be simultaneously active is defined by the kc_nmutex entry in the pSOS+ configuration table.
2. The MU_GLOBAL attribute is ignored by the single-processor version of the pSOS+ kernel.
3. On creation, the mutex is put in the unlocked state.

Multiprocessor Considerations

1. The MU_GLOBAL attribute cannot be specified with MU_PRIO_INHERIT flag.
2. The MU_GLOBAL attribute should be set only if the mutex must be made known to other processor nodes in a multiprocessor configuration. If set, the mutex's name and muid are sent to the master node for entry in its Global Object Table.

3. If the MU_GLOBAL attribute is set and the number of global objects currently exported by the node equals the Microprocessor Configuration Table entry mc_nglbobj, then the mutex is not created and ERR_OBJTFULL is returned.

Callable From

- Task

See Also

mu_delete, *mu_ident*

mu_delete Deletes a mutex.

```
#include <psos.h>
unsigned long mu_delete (
   unsigned long muid    /* MUTEX identifier */
   )
```

Description

This system call deletes a mutex with the specified mutex ID and frees the associated kernel resources. This call can be made only from the node where the specified mutex was created.

The `mu_delete()` system call can be invoked by any task, and not necessarily the task that originally created the mutex. In the event that there are other tasks waiting to acquire that mutex, the tasks are unblocked and an error code is returned to them.

Arguments

`muid`	Specifies the mutex ID of the mutex to be deleted.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A mutex need not be deleted, even when it is no longer in use, except to allow reuse of the associated mutex control block.
2. The calling task may be preempted, if any of the tasks waiting at the deleted mutex has a priority higher than the calling task.
3. Deleting a mutex while another task has it locked could potentially lead to corruption of shared resources. Acquiring the mutex before deleting it will prevent this problem.

4. If a task owned the mutex being deleted, and tasks's priority was raised by the kernel as a result, its priority is recalculated after the deletion of the mutex as if the task had performed a `mu_unlock` operation.

Multiprocessor Considerations

1. This call can be made only from the node where the specified mutex was created.
2. If `muid` identifies a global mutex, `mu_delete()` will notify the master node so that the mutex can be removed from its Global Object Table. Thus, deletion of a global mutex always causes activity on the master node.

Callable From

- Task

See Also

`mu_create`

mu_ident

Obtains the identifier of a named mutex.

```
#include <psos.h>
unsigned long mu_ident(
   char name[4],          /* mutex name */
   unsigned long node,    /* node number */
   unsigned long *muid    /* mutex identifier */
   )
```

Description

This system call enables the calling task to obtain the mutex ID of a mutex it only knows by name. This mutex ID can be used in all other operations relating to the mutex.

The pSOS+ kernel does not check for duplicate names. If multiple mutexes with the same name exist, either on the local node, or on a remote note in a multi-processor system, a `mu_ident()` call may return the ID of any one mutex with the specified name. It can't be ascertained which ID is returned, though, to a certain extent it depends on the standard search order followed by pSOS+ kernel, as defined in the *pSOSystem System Concepts* manual.

Arguments

`name`	Specifies the user assigned name of the mutex.
`node`	Specifies the node selector. For multiprocessing systems, it is a search order specifier. In a single node system, this argument must be 0.
`muid`	Points to the variable where `mu_ident()` stores the ID of the named mutex.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a mutex name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the mutex name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate mutex names. If duplicate mutex names exist, an `mu_ident()` call can return the `muid` of any mutex with the duplicate name.

Multiprocessor Considerations

1. `mu_ident()` converts a mutex's name to its muid by using a search order determined by the `node` input parameter. Because mutexes are created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, then the pSOS+m kernel makes a `mu_ident()` RSC to the master node.

Callable From

- Task

See Also

`mu_create`

mu_info Queries about a Mutex Object.

```
#include <psos.h>
unsigned long mu_info(
    unsigned long muid,             /* Mutex ID */
    struct muinfo *buf,             /* Object Information buffer */
    )
```

Description

This system call provides information of a specified Mutex object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

muid	Specifies the ID of the Mutex object being queried.
buf	Contains the Mutex information.

struct muinfo has the following members.

```
char           name[4];  /* Name of the Mutex */
unsigned long  flags;    /* Mutex attributes */
unsigned long  wqlen;    /* No. of waiting tasks */
unsigned       long      wtid;/* ID of the first waiting task */
unsigned long  count;    /* Mutex reference count */
unsigned long  ownid;    /* ID of mutex owner */
unsigned long  node;     /* Owner task's node number */
unsigned long  ceilprio; /* ceiling priority for MU_PROTECT mutexes */
unsigned long  phpwt;    /* Highest of all waiting task priorities */
```

name specifies the name of mutex object. **flags** has the attributes with which the mutex was created.

wqlen is the number of tasks waiting to lock this mutex. **wtid** is the object ID of the first waiting task. **wtid** is zero if there are no waiting tasks. A **count** value of greater than 0 specifies the mutex is currently locked and the owner is stored in **ownid**. A **count** value of greater than 1 specifies the owner acquired the lock in a recursive fashion. Mutex is unlocked if the **count** is 0. Owner task's node number is returned in **node**.

ceilprio is the ceiling priority of the mutex if the mutex was created with MU_PRIO_PROTECT protocol attribute, otherwise it returns 0. If the mutex was created

with MU_PRIO_INHERIT protocol attribute, **phpwt** represents the priority of the highest priority waiting task, otherwise it returns 0.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The value of the system configuration parameter, SC_PSOS_QUERY, should be set to YES in the file <sys_conf.h>, if the *mu_info*() service call is to be used by the application.

Multiprocessor Considerations

mu_info() can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

mu_create

mu_lock Locks a mutex.

```
#include <psos.h>
unsigned long mu_lock(
   unsigned long muid,   /* mutex identifier */
   unsigned long flags,  /* call attributes */
   unsigned long timeout /* timeout interval */
   )
```

Description

This system call allows a task to lock a mutex specified by the `muid` argument. If the mutex is not locked by another task, the mutex is put in the locked state, and the calling task becomes the owner of the mutex. If the mutex was created with the `MU_RECURSIVE` attribute set (see *mu_create* *on page 2-87*), a task can invoke `mu_lock()` multiple times to acquire the same mutex in a recursive fashion. The same number of `mu_unlock()` calls must then be made by the same task to release the mutex. A mutex created with the `MU_NORECURSIVE` flag can be locked only once by any task. If the task calls `mu_lock` again while it is holding the mutex, the call will return with an error.

If the mutex is locked by another task then, depending on the *flags* argument, the calling task is either put in the blocked state and is queued in the wait queue associated with the mutex, or an error is returned.

Arguments

`muid`	Specifies the mutex ID.	
`flags`	Specifies the flag argument which can take one of the following values:	
	`MU_WAIT`	If the mutex is locked, block until it is available.
	`MU_NOWAIT`	Return with error if the mutex is locked.
`timeout`	Specifies the timeout interval in units of clock ticks. If timeout is zero and `flags` has been set to `MU_WAIT`, the caller will be blocked indefinitely if the mutex is owned by some other task.	

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. When operating on the mutexes, unless the mutex was created with the MU_PRIO_NONE attribute, the pSOS+ kernel may change the priority of the task that owns a mutex to prevent unbounded priority inversion.

Multiprocessor Considerations

If muid identifies a global mutex residing on another processor node, the local kernel will internally make a RSC to that remote node to acquire the mutex. If the MU_WAIT attribute is used, then the pSOS+ kernel on the target node must use an agent to wait for the mutex. If that node is temporarily out of agents, the call will fail. The number of agents on each node is defined by the mc_nagent entry in the node's Multiprocessor Configuration Table.

Callable From

- Task

See Also

mu_unlock

mu_setceil

Optionally gets and changes a mutex's ceiling priority.

```
#include <psos.h>
unsigned long mu_setceil(
   unsigned long muid,      /* mutex identifier */
   unsigned long newprio,   /* new ceiling priority */
   unsigned long *oldprio   /* previous ceiling priority */
   )
```

Description

This system call enables the calling task to optionally obtain and modify a mutex's ceiling priority. The call only operates on mutexes that have been created with the MU_PRIO_PROTECT flag. If oldprio is a non-NULL address, the previous ceiling priority is returned in the variable pointed to by oldprio.

The mu_setceil() call either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex; then it changes the priority ceiling of the mutex and releases the mutex. If the calling task already owns the mutex when it invokes the mu_setceil() call with a valid non-0 value of newprio, then the task's current priority may change according to the priority protect protocol.

Arguments

muid	Specifies the selected mutex for the ceiling priority change.
newprio	Specifies the mutex's new ceiling priority. newprio must be between 1 and 255. If newprio is 0, the mutex's ceiling priority is not changed. This allows a read of a mutex's ceiling priority without changing it.
oldprio	Points to the variable where mu_setceil() stores the mutex's original ceiling priority. If oldprio is a NULL address, the mutex's original ceiling priority is not returned.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

If the calling task's priority gets lowered because of invoking `mu_setceil()` to lower the ceiling priority of a mutex it owns, it can be preempted by a ready task with higher priority. However, it cannot be preempted by a ready task with equal (or lower) priority.

Multiprocessor Considerations

If the `muid` identifies a global mutex residing on another processor node, the local kernel internally makes an RSC to that remote node to change the ceiling priority of the mutex.

Callable From

- Task

See Also

`mu_create`, *`mu_unlock`*

mu_unlock Unlocks a mutex.

```
#include <psos.h>
unsigned long mu_unlock(
   unsigned long muid   /* mutex identifier */
   )
```

Description

This system call allows a task to unlock a mutex specified by the `muid` argument. If the caller currently owns the mutex then `mu_unlock()` call decreases the recursive level associated with the mutex. If the recursive level reaches zero, the mutex is unlocked and the calling task loses ownership of the mutex. An error is returned if the calling task does not currently own the mutex.

For mutexes which have not been created with the `MU_PRIO_PROTECT` flag, when the calling task loses the ownership of the mutex, and if any tasks are waiting to lock the mutex, the ownership of the mutex is transferred to the waiting task at the head of the mutex's wait queue. It is unblocked, and made ready-to-run.

For mutexes which have been created with the `MU_PRIO_PROTECT` flag, when the calling task loses ownership of the mutex, there could be tasks waiting at the mutex's wait queue, on one of two action types. If the task at the head of the queue is waiting to lock the mutex, the ownership of the mutex is transferred to the waiting task, and its priority is raised if necessary. It is unblocked, and made ready-to-run. If the task at the head of the queue is waiting to change the ceiling priority of the mutex, the ceiling priority is modified, the task is unblocked and made ready-to-run; and the next task in the waiting queue is checked. This process of determining the requested action type associated with a waiting task goes on till either the queue is empty, or a task which can successfully lock the mutex is found.

Arguments

`muid` Specifies the mutex ID.

Return Value

This call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The calling task's priority may be lowered as a result of this call. It, however, shall not get preempted by another ready task with equal priority.
2. The calling task may get preempted if the calling task's priority is lowered as a result of this call and/or a higher priority task is woken up as a result of this call. If the woken-up task's priority is raised because of acquiring ownership of the unlocked mutex, it will be scheduled to run before all the ready tasks with equal (or lower) priority.

Multiprocessor Considerations

1. If `muid` identifies a global mutex residing on another processor node, then the pSOS+m kernel will internally make a RSC to that remote node to unlock the mutex.
2. If a task awakened by this call does not reside on the same node as the mutex, the kernel on the mutex's node will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and give it the mutex. Thus, a `mu_unlock()` call, whether it is to a local or remote mutex, may cause pSOS+ activities on another node.

Callable From

- Task

See Also

`mu_lock`, *`mu_create`*, *`mu_setceil`*

ob_roster

Obtains a roster of pSOS+ objects according to a specified type.

```
#include <psos.h>
unsigned long ob_roster(
   unsigned long ob_type,          /* Valid pSOS+ obj type/OT_ALL */
   void *buffer,                   /* Object Roster buffer */
   unsigned long buflen,           /* Length of buffer */
   unsigned long *actlen,          /* Pointer to length returned */
   )
```

Description

This system call obtains a roster of pSOS+ objects, qualified by object type. The information about each object includes minimal information that was specified at object creation time. An array of structures of type **psos_obj_entry** is returned in the user-supplied memory area pointed to by **buffer**, one structure for every object belonging to the roster. If the buffer does not have enough space to hold all the information, only an integral number of structures are placed in the buffer. The location pointed to by **actlen** is updated to contain the actual number of bytes required to store the requested information.

Arguments

ob_type — Specifies the type of the pSOS+ objects whose roster is requested. For a roster of all the objects, a value of OT_ALL is passed. The valid values for this argument are as follows:

OT_TASK	Denotes pSOS+ task object.
OT_QUEUE.	Denotes pSOS+ queue object.
OT_SEM	Denotes pSOS+ semaphore object.
OT_REGION	Denotes pSOS+ region object.
OT_PART	Denotes pSOS+ partition object.
OT_MUTEX	Denotes pSOS+ mutex object.
OT_CVAR	Denotes pSOS+ condition variable object.
OT_TSD	Denotes pSOS+ task-specific data object.
OT_ALL	Denotes pSOS+ objects of all types.

`buffer`	User-supplied buffer address, to store the roster information.
`buflen`	Contains the length of the user-supplied buffer.
`actlen`	Contains pointer to the location which is updated to store the actual number of bytes of information returned in the buffer.

struct psos_obj_entry has the following members.

```
char              name[4];      /* Name of the object */
unsigned long     id;           /* Object ID*/
unsigned long     type;         /* Type of object*/
```

name specifies the name, **id** the object ID, and **type** the object type, of the pSOS+ object.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The value of the `system configuration parameter, SC_PSOS_QUERY, should be set to YES` in the file `<sys_conf.h>` if the `ob_roster()` service call is to be used by the application.

Multiprocessor Considerations

`ob_roster()` returns only the objects that are accessible from the Local and Global Object tables resident on the local node. On a slave node, the roster is of objects (local and global) created on that node. However, `ob_roster()` calls executed on the master node will return a roster of all the objects created on the node, as well as all the global objects created elsewhere in the system.

Callable From

- Task
- Callout

See Also

cv_info, *mu_info*, *pt_info*, *q_info*, *q_vinfo*, *rn_info*, *sm_info*, *t_info*, *tsd_info*,

pt_create

Creates a memory partition of fixed-size buffers.

```
#include <psos.h>
unsigned long pt_create(
   char name[4],          /* partition name */
   void *paddr,           /* partition physical addr. */
   void *laddr,           /* partition logical address */
   unsigned long length,  /* partition length in bytes */
   unsigned long bsize,   /* buffer size in bytes */
   unsigned long flags,   /* buffer attributes */
   unsigned long *ptid,   /* partition identifier */
   unsigned long *nbuf    /* number of buffers created */
   )
```


Description

This service call enables a task to create a new memory partition, from which fixed-sized memory buffers can be allocated for use by the application. The pSOS+ kernel takes a portion from the top of this region to use as its Partition Control Block.

Arguments

`name` Specifies the user-assigned name for the new partition.

`paddr` Specifies the physical memory address of the partition.

`laddr` Specifies the logical address of the partition generated after MMU-translation; `laddr` is ignored on non-MMU systems.

`length` Specifies the total partition length in bytes.

`bsize` Specifies the size of the buffers. `bsize` must be a power of 2, and equal to or greater than 4.

`flags` Specifies the attributes of the buffer. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that a partition is globally addressable, you place the symbolic constant `PT_GLOBAL` in `flags`. To specify that the partition is globally addressable and that it prohibits deletion with outstanding buffers, you place both `PT_GLOBAL` and `PT_NODEL` in `flags`, using the following syntax:

```
PT_GLOBAL| PT_NODEL
```

	PT_GLOBAL	Partition is globally addressable by other nodes. The single-processor version of the pSOS+ kernel ignores PT_GLOBAL.
	PT_LOCAL	Partition can be addressed only by the local node.
	PT_DEL	Deletion of the partition with pt_delete() is enabled, even if one or more buffers are allocated.
	PT_NODEL	Deletion of the partition is prohibited unless all buffers have been freed.
ptid		Points to the variable where pt_create() stores the partition ID of the named partition.
nbuf		Points to the variable where pt_create() stores the number of actual buffers in the partition.

Return Value

This call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a partition name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the partition name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate partition names. If duplicate names exist, a pt_ident() call can return the ptid of any partition with the duplicate name.

Multiprocessor Considerations

1. The PT_GLOBAL attribute should be set only if the partition must be made known to other processor nodes in a multiprocessor configuration. If set, the partition's name and ptid are sent to the master node for entry in the Global Object Table.
2. If the PT_GLOBAL attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry mc_nglbobjs, the partition is not created and ERR_OBJTFULL is returned.

Callable From

- Task

See Also

pt_ident, *pt_getbuf*

pt_delete Deletes a memory partition.

```
#include <psos.h>
unsigned long pt_delete (
   unsigned long ptid    /* partition identifier */
   )
```

Description

This system call deletes a memory partition specified by its ID. Unless the PT_DEL attribute was specified when the partition was created, `pt_delete()` returns an error if any buffers allocated from the partition have not been returned.

The calling task does not have to be the creator (parent) of the partition to be deleted. However, a partition must be deleted from the node on which it was created.

Arguments

`ptid`	Specifies the partition identifier.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Once created, a partition is generally used by multiple tasks for data buffers, which can be passed around between tasks, or even between nodes. There is rarely a reason for deleting a partition, even when it is no longer used, except to allow reuse of memory occupied by the partition.

Multiprocessor Considerations

If `ptid` identifies a global partition, `pt_delete` notifies the master node so the partition can be removed from its Global Object Table. Thus, deletion of a global partition always causes activity on the master node.

Callable From

- Task

See Also

pt_create

pt_getbuf

Gets a buffer from a partition.

```
#include <psos.h>
unsigned long pt_getbuf(
   unsigned long ptid,   /* partition identifier */
   void **bufaddr        /* starting address of buffer */
   )
```

Description

This system call gets a buffer from a partition. If the partition is empty, an error is returned.

Arguments

`ptid`	Specifies the partition identifier.
`bufaddr`	Points to the variable where `pt_getbuf()` stores the starting address of the allocated buffer.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Buffers always start on long word boundaries.
2. It is not possible to wait for a buffer. `pt_getbuf()` unconditionally returns.

Multiprocessor Considerations

If the input `ptid` identifies a global partition residing on another processor node, then the pSOS+ kernel internally makes an RSC to that remote node to allocate the buffer.

Callable From

- Task.
- ISR, if the partition is local to the node from which `pt_getbuf()` is made.
- KI, if the partition is local to the node from which `pt_getbuf()` is made.
- Callout, if the partition is local to the node from which `pt_getbuf()` is made.

See Also

`pt_retbuf`

pt_ident

Obtains the identifier of a named partition.

```
#include <psos.h>
unsigned long pt_ident(
   char name[4],          /* partition name */
   unsigned long node,    /* node number */
   unsigned long *ptid    /* partition identifier */
   )
```

Description

This system call enables the calling task to obtain the partition ID of a memory partition it only knows by name. This partition ID can be used in all other operations relating to the memory partition.

Most system calls, except `pt_create()` and `pt_ident()`, reference a partition by its partition ID. `pt_create()` returns the partition ID to the partition creator. For other tasks, one way to obtain the partition ID is to use `pt_ident()`.

Arguments

`name`	Specifies the name of the partition.
`node`	For multiprocessing systems, is a search order specifier. See *Multiprocessor Considerations*. In a single node system, this argument must be 0.
`ptid`	Points to the variable where `pt_ident()` stores the ID of the named partition.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a partition name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the partition name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate partition names. If duplicate partition names exist, a `pt_ident()` call can return the ID of any partition with the duplicate name.

Multiprocessor Considerations

1. `pt_ident()` converts a partition's name to its `ptid` using a search order determined by the `node` input parameter, which is described in *pSOSystem System Concepts*. Because partitions created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, then the pSOS+m kernel makes an RSC to the master node.

Callable From

- Task

See Also

pt_create

pt_info Queries a Partition Object.

```
#include <psos.h>
unsigned long pt_info(
    unsigned long ptid,             /* Partition ID */
    struct ptinfo *buf,             /* Object Information buffer */
    )
```

Description

This system call returns information of a specified Partition object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

ptid	Specifies the ID of the Partition object being queried.
buf	Contains the Partition information.

struct ptinfo has the following members.

```
char           name[4];  /* Name of the partition */
unsigned long  flags;    /* Partition attributes */
unsigned long  bsize;    /* Size of a partition buffer */
unsigned long  nbufs;    /* Total no. of buffers in the partition */
unsigned long  nfree;    /* Total no. of free buffers */
void           *iaddr;   /* Partition start address */
unsigned long  length;   /* Total length of partition in bytes */
```

name returns the partition name. The attributes with which this partition was created are in **flags**. The buffer size of this partition is in **bsize**. The total length of the partition is returned in **length**.

The total number of buffers and the free buffers in this partition are given by **nbufs** and **nfree** respectively.

The internal start address of this partition is in **iaddr**.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, SC_PSOS_QUERY, to YES in the file <sys_conf.h> to access the pSOS+ pt_info() service call.

Multiprocessor Considerations

- pt_info() can be called only on objects created on the local node.
- m_int2ext() returns the external address corresponding a particular internal address.

Callable From

- Task
- Callout

See Also

pt_create, *m_int2ext*

pt_retbuf Returns a buffer to the partition from which it came.

```
#include <psos.h>
unsigned long pt_retbuf(
   unsigned long ptid,   /* partition identifier */
   void *bufaddr         /* starting address of the buffer */
   )
```

Description

This system call returns a buffer to the partition from which it was allocated. Because the pSOS+ kernel does not keep track of buffer ownership, it is possible for one task to get a buffer, and another task to return it.

Arguments

`ptid`	Specifies the partition ID of the buffer to return.
`bufaddr`	Specifies the buffer's starting address.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

If the input `ptid` identifies a global partition residing on another processor node, then the pSOS+ kernel internally makes an RSC to that remote node to return the buffer.

Callable From

- Task.

- ISR, if the partition is local to the node from which the `pt_retbuf()` call is made.
- KI, if the partition is local to the node from which the `pt_retbuf()` call is made.
- Callout, if the partition is local to the node from which the `pt_retbuf()` call is made.

See Also

`pt_getbuf`

pt_sgetbuf Gets a buffer from a partition.

```
#include <psos.h>
unsigned long pt_sgetbuf(
   unsigned long ptid,   /* partition identifier */
   void **paddr,   /* physical address */
   void **laddr    /* logical address */
   )
```

Description

This system call gets a buffer from a partition. If the partition is empty, an error is returned.

On MMU-based systems, both physical and logical addresses are returned to simplify transfer of buffers between supervisor and user mode programs. In non-MMU systems, the logical address is the same as the physical address, and this call functions the same as the `pt_getbuf()` call.

This service is available in the non-MMU versions of the pSOS+ kernel for the sole purpose of enabling software designed for MMU-based systems to run, unmodified, on systems without MMU.

Arguments

`ptid`	Specifies the buffer's partition ID.
`paddr`	Points to the variable where `pt_sgetbuf()` stores the physical address of the buffer.
`laddr`	Points to the variable where `pt_sgetbuf()` stores the logical address of the buffer.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Buffers always start on long word boundaries.
2. It is not possible to wait for a buffer. `pt_sgetbuf()` unconditionally returns.

Multiprocessor Considerations

If the input argument `ptid` identifies a global partition on another processor node, the pSOS+ kernel internally makes an RSC to that remote node to allocate the buffer.

Callable From

- Task

See Also

`pt_retbuf`, *`pt_getbuf`*

q_asend

(pSOS+m kernel only) Asynchronously posts a message to an ordinary message queue.

```
#include <psos.h>
unsigned long q_asend(
   unsigned long qid,          /* queue identifier */
   unsigned long msg_buf[4]    /* message buffer */
   )
```

Description

This system call functions the same as `q_send()` except that it executes asynchronously. Refer to the description of `q_send()` for further information. For a detailed description of asynchronous services, refer to the *pSOSystem Systems Concepts* manual.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Specifies the message to send.

Return Value

When called in a system running the pSOS+m kernel this call always returns 0. The pSOS+ kernel (the single processor version) returns `ERR_SSFN`.

Error Codes

Should the call fail, if present, the node's `MC_ASYNCERR` routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the `MC_ASYNCERR` routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+m kernel.
2. The calling task can be preempted as a result of this call.

3. `q_asend()` asynchronously sends a message to an ordinary message queue. Use `q_avsend()` to send a message asynchronously to a variable length message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to post the input message to that queue.

2. If a task awakened by this call does not reside on the local node, then the pSOS+m kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_asend()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task

See Also

q_send, *q_vsend*, *q_avsend*, *q_aurgent*, *q_receive*, *q_notify*

q_aurgent

(pSOS+m kernel only) Asynchronously posts a message at the head of a variable-length message queue.

```
#include <psos.h>
unsigned long q_aurgent(
   unsigned long qid,          /* queue identifier */
   unsigned long msg_buf[4]    /* message buffer */
   )
```

Description

This system call functions the same as the `q_urgent()` call except that it executes asynchronously. Refer to the description of `q_urgent()` for further information.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Specifies the message to send.

Return Value

When called in a system running the pSOS+m kernel, this call always returns 0. The pSOS+ kernel (the single processor version) returns ERR_SSFN.

Error Codes

Should the call fail, if present, the node's MC_ASYNCERR routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the MC_ASYNCERR routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+m kernel.
2. The calling task can be preempted as a result of this call.
3. `q_aurgent()` asynchronously sends an urgent message to an ordinary message queue. Use `q_avurgent()` to asynchronously send an urgent message to a variable length message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to post the input message to that queue.
2. If a task awakened by this call does not reside on the local node, then the pSOS+m kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_aurgent()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task

See Also

`q_urgent`, *`q_vurgent`*, *`q_avurgent`*, *`q_asend`*, *`q_receive`*, *`q_notify`*

q_avsend

(pSOS+m kernel only) Asynchronously posts a message to a variable-length message queue.

```
#include <psos.h>
unsigned long q_avsend(
   unsigned long qid,       /* queue identifier */
   void *msg_buf,           /* message buffer */
   unsigned long msg_len,   /* length of message */
   )
```

Description

This system call functions the same as the q_vsend() call except that it executes asynchronously. Refer to the description of q_vsend() for further information.

Arguments

qid	Specifies the queue ID of the target queue.
msg_buf	Points to the message to send.
msg_len	Specifies the length of the message. It must not exceed the queue's maximum message length.

Return Value

When called in a system running the pSOS+m kernel, this call always returns 0. The pSOS+ kernel (the single processor version) returns ERR_SSFN.

Error Codes

Should the call fail, if present, the node's MC_ASYNCERR routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the MC_ASYNCERR routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+m kernel.
2. The calling task can be preempted as a result of this call.

3. The pSOS+m kernel must copy the message into a queue buffer or the receiving task's buffer. Longer messages take longer to copy. Users should account for the copy time in their designs.

4. `q_avsend()` asynchronously sends a message to a variable length message queue. Use `q_asend()` to send a message asynchronously to an ordinary message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to post the input message to that queue.

2. If a task awakened by this call does not reside on the local node, the local kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_avsend()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task

See Also

`q_vsend`, *`q_send`*, *`q_asend`*, *`q_urgent`*, *`q_vreceive`*, *`q_vnotify`*

q_avurgent

(pSOS+m kernel only) Asynchronously posts a message at the head of a variable-length message queue.

```
#include <psos.h>
unsigned long q_avurgent(
   unsigned long qid,      /* queue identifier */
   void *msg_buf,          /* message buffer */
   unsigned long msg_len   /* length of message */
   )
```

Description

This system call functions the same as q_vurgent except that q_avurgent executes asynchronously. Refer to the description of q_vurgent for further information. For a more detailed description of asynchronous services, refer to the *pSOSystem System Concepts* manual.

Arguments

qid	Specifies the queue identifier.
msg_buf	Points to the message to send.
msg_len	Specifies the length of the message.

Return Value

When called in system running pSOS+m, this call always returns 0. The pSOS+ kernel (the single processor version) returns ERR_SSFN.

Error Codes

Should the call fail, if present, the node's MC_ASYNCERR routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the MC_ASYNCERR routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

1. This call is supported only by the pSOS+m kernel.
2. The calling task can be preempted as a result of this call.
3. The pSOS+m kernel must copy the message into a queue buffer or the receiving task's buffer. Longer messages take longer to copy. Users should account for the copy time in their designs.
4. `q_avsend()` asynchronously sends a message to a variable length message queue. Use `q_asend()` to asynchronously send a message to an ordinary message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to post the input message to that queue.
2. If a task awakened by this call does not reside on the local node, the local kernel internally passes the message to the task's node of residence, whose pSOS+m kernel readies the task and gives it the relayed message. Thus, a `q_avurgent()` call, whether it is on the local or a remote queue, can cause pSOS+m activity on another processor node.

Callable From

- Task

See Also

`q_urgent`, *`q_vurgent`*, *`q_vreceive`*, *`q_vsend`*, *`q_vnotify`*

q_broadcast

Broadcasts identical messages to an ordinary message queue.

```
#include <psos.h>
unsigned long q_broadcast(
   unsigned long qid,        /* queue identifier */
   unsigned long msg_buf[4], /* msg. of 4 long words */
   unsigned long *count      /* number of tasks */
   )
```

Description

This system call enables the caller to wake up all tasks that might be waiting at an ordinary message queue. If the task queue is empty, this call does nothing. If one or more tasks are waiting at the queue, `q_broadcast()` gives a copy of the input message to each such task and makes it ready to run. After a `q_broadcast()` call, no tasks will be waiting to receive a message from the specified queue.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Specifies the message to send.
`count`	Points to the variable where `q_broadcast()` stores the number of tasks readied by the broadcast.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `q_broadcast()` is particularly useful in situations where a single event (for example, an interrupt) must wake up more than one task. In such cases, `q_broadcast()` is clearly more efficient than multiple `q_send()` calls.
2. If the caller is a task, it may be preempted as a result of this call.

3. `q_broadcast()` can be intermixed with `q_send()` and `q_urgent()` calls to the same queue.

4. `q_broadcast()` sends messages to an ordinary message queue. Use `q_vbroadcast()` to send messages to a variable length message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, then the pSOS+m kernel will internally make an RSC to that remote node to post the input message to that queue.

2. If tasks awakened by this call do not reside on the local node, then the pSOS+m kernel will internally pass the message to each task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_broadcast()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on one or more other processor nodes.

Callable From

- Task.
- ISR, if the targeted queue is local to the node from which the `q_broadcast()` call is made.
- KI, if the targeted queue is local to the node from which the `q_broadcast()` call is made.
- Callout, if the targeted queue is local to the node from which the `q_broadcast()` call is made.

See Also

`q_send`, *`q_receive`*, *`q_vbroadcast`*

q_create

Creates an ordinary message queue.

```
#include <psos.h>
unsigned long q_create(
   char name[4],         /* queue name */
   unsigned long count,  /* queue size */
   unsigned long flags,  /* queue attributes */
   unsigned long *qid    /* queue identifier */
   )
```

Description

This system call creates an ordinary message queue by allocating and initializing a Queue Control Block (QCB) according to the specifications supplied with the call.

Like all objects, a queue has a user-assigned name and a pSOS-assigned queue ID returned by `q_create()`. Several flag bits specify the characteristics of the message queue. Tasks can wait for messages either by task priority or strictly FIFO, and a limit can be optionally set on the maximum number of messages that can be simultaneously posted at the queue.

Arguments

`name` Specifies the user-assigned name of the new message queue.

`count` If `Q_LIMIT` is set (see *flags*, below), then the `count` argument specifies the maximum number of messages that can be simultaneously posted at the queue. If `Q_PRIBUF` is also set, then the argument `count` also specifies the number of buffers set aside from the system-wide pool of message buffers for the private use of this queue. If `Q_NOLIMIT` is set, `count` is ignored.

`flags` Specifies the attributes of the queue. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that the queue is globally addressable, you place `Q_GLOBAL` in `flags`. To specify that the queue is globally addressable and that tasks are queued by FIFO, you place `Q_GLOBAL` and `Q_FIFO` in `flags`, using the following syntax:

```
Q_GLOBAL | Q_FIFO
```

`Q_GLOBAL`	Queue is globally addressable by other nodes.
`Q_LOCAL`	Queue is addressable only by the local node.

Q_PRIOR	Tasks are queued by priority.
Q_FIFO	Tasks are queued by FIFO.
Q_LIMIT	Message queue size is limited to count.
Q_NOLIMIT	Message queue size is unlimited.
Q_PRIBUF	Private buffers are allocated for message storage.
Q_SYSBUF	System buffers are used for message storage.

qid — Points to the variable where q_create() stores the queue ID of the named queue.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a queue name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the queue name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate queue names. If duplicate names exist, a q_ident() call can return the qid of any queue with the duplicate name.
3. The maximum number of queues that can be simultaneously active is defined by the kc_nqueue entry in the pSOS+ Configuration Table. The count argument is ignored if the Q_NOLIMIT attribute is specified.
4. A queue created with Q_NOLIMIT specified is slightly more efficient.
5. Q_LIMIT and a count equal 0 is a legitimate setting. This combination has the interesting property that a q_send() will succeed only if there is already a task waiting; otherwise, q_send() will fail.
6. Q_LIMIT set with Q_PRIBUF guarantees that enough buffers will be available for messages to be posted at this queue. If Q_LIMIT is not set, then Q_PRIBUF is ignored.

7. If a queue is created without private buffers, then messages posted to it will be stored in buffers from the system-wide pool on the node where the queue resides. The size of this pool is defined by the `kc_nmsgbuf` entry in the node's pSOS+ Configuration Table.

8. The `Q_GLOBAL` attribute is ignored by the single-processor version of the pSOS+ kernel.

9. `q_create()` creates an ordinary message queue. Use `q_vcreate()` to create a variable length message queue.

Multiprocessor Considerations

1. The `Q_GLOBAL` attribute should be set only if the queue must be made known to other processor nodes in a multiprocessor configuration. If set, the queue's `name` and `qid` are sent to the master node for entry in its Global Object Table.

2. If the `Q_GLOBAL` attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry `mc_nglbobj`, then the queue is not created and `ERR_OBJTFULL` is returned.

Callable From

- Task

See Also

`q_ident`, *`q_delete`*, *`q_vcreate`*

q_delete

Deletes an ordinary message queue.

```
#include <psos.h>
unsigned long q_delete(
   unsigned long qid    /* queue identifier */
   )
```


Description

This system call deletes the ordinary message queue with the specified queue ID, and frees the QCB. `q_delete()` takes care of cleaning up the queue. If there are tasks waiting, they will be unblocked and given an error code. If some messages are queued there, the message buffers, along with any free private buffers are returned to the system-wide pool.

The calling task does not have to be the creator of the queue in order to be deleted. However, a queue must be deleted from the node on which it was created.

Arguments

`qid` Specifies the queue ID of the queue to delete.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Once created, a queue is generally used by multiple tasks for communication and synchronization. There is rarely a reason for deleting a queue, even when it is no longer used, except to allow reuse of the QCB.
2. The calling task may be preempted after this call, if a task that is waiting for a message from the deleted queue has higher priority.
3. Any pending messages are lost.

4. `q_delete()` deletes an ordinary message queue. Use `q_vdelete()` to delete a variable length message queue.

Multiprocessor Considerations

If `qid` identifies a global queue, `q_delete` will notify the master node so that the queue can be removed from its Global Object Table. Thus, deletion of a global queue always causes activity on the master node.

Callable From

- Task

See Also

q_create, *q_vdelete*

q_ident

Obtains the queue ID of an ordinary message queue.

```
#include <psos.h>
unsigned long q_ident(
   char name[4],          /* queue name */
   unsigned long node,    /* node number */
   unsigned long *qid     /* queue identifier */
   )
```


Description

The intended purpose of this system call is to enable the calling task to obtain the queue ID of an ordinary message queue. However, since a variable length message queue is just a special type of message queue, `q_ident()` and `q_vident()` are functionally identical. Both return the queue ID of the first queue encountered with the specified name, whether it be ordinary or variable length.

Most system calls, except `q_create()`/`q_vcreate()` and `q_ident()`/`q_vident()`, reference a queue by its queue ID. For other tasks, one way to obtain the queue ID is to use `q_ident()`/`q_vident()`. Once obtained, the queue ID can then be used in all other operations relating to this queue.

Arguments

`name`	Specifies the name of the message queue.
`node`	For multiprocessing systems, is a search order specifier. See *Multiprocessor Considerations*. In a single node system, this argument must be 0.
`qid`	Points to the variable where `q_ident()` stores the ID of the named message queue.

Return Value

The system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a queue name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the queue name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate queue names. If duplicate names exist, a `q_ident()` call can return the `qid` of any queue with the duplicate name.

Multiprocessor Considerations

1. `q_ident()` converts a queue's name to its qid using a search order determined by the `node` input parameter, as described in *pSOSystem System Concepts*. Because queues created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, the local kernel makes a `q_ident()` RSC to the master node.

Callable From

- Task

See Also

`q_create`, *`q_vident`*

q_info

Queries a Message Queue Object.

```
#include <psos.h>
unsigned long q_info(
   unsigned long qid,              /* Message Queue ID */
   struct qinfo *buf,              /* Object Information buffer */
   )
```


Description

This system call returns information of a fixed length message queue object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

qid	Specifies the ID of the Message Queue object being queried.
buf	Contains the Message Queue information.

struct qinfo has the following members.

```
char           name[4];  /* Name of the queue */
unsigned long  flags;    /* Queue attributes */
unsigned long  wqlen;    /* No. of waiting tasks */
unsigned long  wtid;     /* ID of the first waiting task */
unsigned long  mqlen;    /* No. of messages in the queue */
unsigned long  mqmax;    /* Max no. of messages allowed */
unsigned long  tid_ntfy; /* Task to be notified of message arrival */
unsigned long  ev_ntfy;  /* Events to post to notify mesg arrival */
```

name of the queue is returned in **name**.

flags returns the options with which the queue was created. **wqlen** gives the number of tasks waiting on this queue to receive messages. **wtid** stores the ID of the first waiting task. **wtid** is zero when there are no waiting tasks.

The value in **mqlen** is the number of messages in the queue to be picked up. The value in **mqmax** is the max. number of messages allowed for this queue. If the queue was created with Q_NOLIMIT flag, **mqmax** returns 0.

If the queue reserves private buffers, the number of such buffers still available any time are **mqmax - mqlen**.

Whenever a message is posted onto the queue, the task denoted by **tid_ntfy**, is notified by the sending of an event **ev_ntfy**. A 0 value for **tid_ntfy** denotes that there is no task to notify on message arrival; and a 0 value for **ev_ntfy** denotes that there is no event to post.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, `SC_PSOS_QUERY`, to `YES` in the file `<sys_conf.h>` to access the pSOS+ `q_info()` service call.

Multiprocessor Considerations

`q_info()` can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

`q_create`, *`q_send`*

q_notify

Registers the events for notification of availability of a message.

```
#include <psos.h>
unsigned long q_notify(
   unsigned long qid,              /* ID of target queue */
   unsigned long tid,              /* ID of task to be notified */
   unsigned long events            /* bit-encoded events */
   )
```


Description

This system call registers a set of bit-encoded events and a task ID for the given message queue. Once so registered, a notification is sent to the task whenever a message is posted to the queue, via an implicit call of the form: `ev_send(tid, events)`. The task can choose to receive the notification via `ev_receive()` call.

Arguments

`qid`	Specifies the ID of the message queue for which the notification of availability of message is to be registered.
`tid`	Specifies the ID of the task that must be notified of the availability of the message. A value of 0 registers the calling task as the one to be notified.
`events`	Contains a list of bit-encoded events. A value of 0 turns off the event notification mechanism.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A message can be posted to a message queue, and hence the notification event can be generated only as a result of `q_send()` or `q_urgent()` calls. Note that

the q_broadcast() call never posts a message to a queue; it only delivers the message to any tasks waiting to receive the message.

2. When event notification for availability of a message is requested, the notification is generated only when there are no tasks waiting to receive a message. If there are tasks waiting to receive a message, one such task is awakened by pSOS+ and given the message but no notification is generated.

3. The ID of the task specified by tid argument is not verified during q_notify() call, but it is verified every time a notification is sent.

4. A notification does not guarantee that the message will remain available to be grabbed by the task that was notified. A higher priority task, or even an ISR, can potentially obtain the message before the notified task has a chance to run.

5. This call can be used to register a notification for a variable length message queue, since variable length message queues are a special type of message queue. However, for the sake of consistency, a separate call named q_vnotify() has been provided that has the same functionality as q_notify().

Multiprocessor Considerations

None, q_notify() can be called only from the local node.

Callable From

- Task

See Also

q_send, *q_urgent*, *q_vnotify*, *sm_notify*, *as_notify*, *ev_send*, *ev_receive*

q_receive

Requests a message from an ordinary message queue.

```
#include <psos.h>
unsigned long q_receive(
   unsigned long qid,       /* queue identifier */
   unsigned long flags,     /* queue attributes */
   unsigned long timeout,   /* timeout in clock ticks */
   unsigned long msg_buf[4] /* message buffer */
   )
```


Description

This system call enables a task or an ISR to obtain or examine a message at the head of an ordinary message queue.

The caller can examine the contents of the message at the head of the queue by specifying the Q_PEEK flag. If there are messages in the queue, the contents of the message at the head of the queue is returned, and the message is left pending in the queue. If there are no messages available in the queue, the system returns an error, no matter whether the caller specified Q_WAIT or Q_NOWAIT.

The caller can dequeue the message at the head of the queue by specifying the Q_DEQUEUE flag. When this flag has been specified, the behavior is as follows:

If the queue is non-empty, this call dequeues and returns the first message there. If the queue is empty and the caller specified Q_NOWAIT, q_receive() returns with an error code. If Q_WAIT is selected, the caller will be blocked until a message is posted to the queue, or if the timeout argument is used, until the timeout occurs, whichever happens first. If timeout is zero and Q_WAIT is selected, then q_receive() will wait forever. The timeout argument is ignored if Q_NOWAIT is selected.

Arguments

qid	Specifies the queue ID of the target queue.
flags	Specifies whether q_receive() will block waiting for a message. flags is formed by ORing the following symbolic constants (one from each pair), which are defined in <psos.h>:
	Q_NOWAIT Don't wait for message.
	Q_WAIT Wait for message.

	Q_DEQUEUE	Dequeue and return the message at the head of the queue.
	Q_PEEK	Return the message at the head of the queue without dequeuing it.
timeout	Specifies the timeout interval, in units of clock ticks.	
msg_buf	An output parameter. Contains the received message.	

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If it is necessary to block the calling task, q_receive() will enter the calling task at the message queue's task-wait queue. If the queue was created with the Q_FIFO attribute, then the caller is simply entered at the tail of the wait queue. If the queue was created with the Q_PRIOR attribute, then the task will be inserted into the wait queue by priority.
2. q_receive() requests a message from an ordinary message queue. Use q_vreceive() to request a message from a variable length message queue.

Multiprocessor Considerations

If qid identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to request a message from that queue. If the Q_WAIT attribute is elected, then the pSOS+m kernel on the target node must use an *agent* to wait for the message. An agent is an internal object created by pSOS+ to simulate a task on a remote node. If the node is temporarily out of agents, the call will fail. The number of agents on each node is defined by the mc_nagent entry in the Multiprocessor Configuration Table.

Callable From

- Task.

- ISR, if Q_NOWAIT is set and the target queue is local to the node from which the q_receive call is made.
- KI, if Q_NOWAIT is set and the target queue is local to the node from which the q_receive call is made.
- Callout, if Q_NOWAIT is set and the target queue is local to the node from which the q_receive call is made.

See Also

q_send, *q_vreceive*, *q_notify*

q_send

Posts a message to an ordinary message queue.

```
#include <psos.h>
unsigned long q_send(
   unsigned long qid,          /* queue identifier */
   unsigned long msg_buf[4]    /* message buffer */
   )
```

Description

This system call is used to send a message to a specified ordinary message queue. If a task is already waiting at the queue, the message is passed to that task, which is then unblocked and made ready to run. If no task is waiting, the input message is copied into a message buffer from the system pool or, if the queue has private buffers, into a private message buffer, which is then put in the message queue behind any messages already posted to the queue.

If, as a result of this call, a message becomes pending in the queue, and a set of task and events had been registered via a prior call to `q_vnotify()` to notify the availability of messages in the queue, pSOS+ also performs an `ev_send()` operation to send the registered events to the registered task.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Specifies the message to send.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the caller is a task, it may be preempted as a result of this call.
2. `q_send()` sends a message to an ordinary message queue. Use `q_vsend()` to send a message to a variable length message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to post the input message to that queue.
2. If a task awakened by this call does not reside on the local node, the local kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_send()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task.
- ISR, if the target queue is local to the node from which the `q_send()` call is made.
- KI, if the target queue is local to the node from which the `q_send()` call is made.
- Callout, if the target queue is local to the node from which the `q_send()` call is made.

See Also

`q_broadcast`, *`q_receive`*, *`q_urgent`*, *`q_vsend`*, *`q_notify`*

q_urgent

Posts a message at the head of an ordinary message queue.

```
#include <psos.h>
unsigned long q_urgent(
   unsigned long qid,          /* queue identifier */
   unsigned long msg_buf[4]    /* message buffer */
   )
```

Description

This system call is identical in all respects to `q_send()` with one exception: if one or more messages are already posted at the target queue, then the new message will be inserted into the message queue in front of all such queued messages.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Specifies the message to send.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `q_urgent()` is useful when the message represents an urgent errand and must be serviced ahead of the normally FIFO ordered messages.
2. If the caller is a task, it may be preempted as a result of this call.
3. `q_urgent()` sends a message to an ordinary message queue. Use `q_vurgent()` to send a message to a variable length message queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, the local kernel internally makes an RSC to that remote node to post the input message to that queue.
2. If a task awakened by this call does not reside on the local node, the local kernel internally passes the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_urgent()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task.
- ISR, if the target queue is local to the node from which the `q_urgent()` call is made.
- KI, if the target queue is local to the node from which the `q_urgent()` call is made.
- Callout, if the target queue is local to the node from which the `q_urgent()` call is made.

See Also

`q_receive`, *`q_send`*, *`q_vurgent`*, *`q_notify`*

q_vbroadcast Broadcasts identical variable-length messages to a message queue.

```
#include <psos.h>
unsigned long q_vbroadcast(
   unsigned long qid,     /* queue identifier */
   void *msg_buf,         /* message buffer */
   unsigned long msg_len, /* length of message */
   unsigned long *count   /* number of tasks */
   )
```

Description

This system call sends a message to all tasks waiting at a specified variable length queue. Otherwise, it is identical to `q_broadcast()`.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Points to the message to send.
`msg_len`	Specifies the length of the message. It must not exceed the queue's maximum message length, which was specified with `q_vcreate()`.
`count`	Points to the variable where `q_vbroadcast()` stores the number of tasks readied by the broadcast.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the caller is a task, it may be preempted as a result of this call.
2. `q_vbroadcast()` can be intermixed with `q_vsend()` and `q_vurgent()` calls to the same queue.

3. The pSOS+ kernel must copy the message from the caller's buffer to a receiving task's buffer. Longer messages take longer to copy. Users should account for the copy time in their designs, especially when calling from an ISR.

4. `q_vbroadcast()` sends messages to a variable length message queue. Use `q_broadcast()` to send messages to an ordinary queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to post the input message to that queue.

2. If tasks awakened by this call do not reside on the local node, the local kernel internally passes the message to each task's node of residence, whose pSOS+ kernel will ready the task and give it the relayed message. Thus, a `q_vbroadcast()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on one or more processor nodes.

Callable From

- Task.
- ISR, if the target queue is local to the node from which the `q_vbroadcast()` call is made.

See Also

`q_broadcast`, *`q_vsend`*, *`q_vreceive`*

q_vcreate

Creates a variable-length message queue.

```
#include <psos.h>
unsigned long q_vcreate(
   char name[4],          /* queue name */
   unsigned long flags,   /* queue characteristics */
   unsigned long maxnum,  /* maximum number of messages */
   unsigned long maxlen,  /* maximum message length */
   unsigned long *qid     /* queue identifier */
   )
```

Description

This system call creates a queue that supports variable length messages. Otherwise, it is identical to `q_create()`. `q_vcreate` creates a message queue by allocating and initializing a Queue Control Block (QCB) according to the specifications supplied with the call.

Arguments

`name` Specifies the user-assigned name of the new message queue.

`flags` Specifies the attributes of the queue. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that the queue is globally addressable, you place `Q_GLOBAL` in `flags`. To specify that the queue is globally addressable and that tasks are queued by FIFO, you place `Q_GLOBAL` and `Q_FIFO` in `flags`, using the following syntax:

```
Q_GLOBAL | Q_FIFO
```

`Q_GLOBAL`	Queue is globally addressable by other nodes.
`Q_LOCAL`	Queue is addressable only by the local node.
`Q_PRIOR`	Tasks are queued by priority.
`Q_FIFO`	Tasks are queued by FIFO.

`maxnum` Specifies the maximum number of messages that can be pending at one time at the queue.

`maxlen` Specifies the maximum message size (in bytes).

`qid` Points to the variable where `q_vcreate()` stores the queue's pSOS-assigned queue ID.

Queues created by q_vcreate() always have a fixed number of private buffers. The pSOS+ kernel uses maxnum and maxlen to allocate sufficient memory for message storage from region 0. If insufficient memory is available, an error is returned. Queues created with q_vcreate() never allocate or use message buffers from the pSOS+ message buffer pool.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a queue name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the queue name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate queue names. If duplicate names exist, an q_vident() can return the qid of any queue with the duplicate name.
3. The maximum number of queues that can be simultaneously active is defined by the entry kc_nqcb in the pSOS+ Configuration Table. It applies to the combined total of both fixed and variable queues.
4. The Q_GLOBAL attribute is ignored by the single-processor version of the pSOS+ kernel.
5. A special case occurs when maxnum is set to 0. In this case, a message can only be successfully sent if a task is already waiting at the queue.
6. A special case occurs when maxlen is set to 0. In this case, the queue behaves much like a counting semaphore.
7. No memory is allocated by the queue when either maxlen or maxnum is set to 0. The amount of Region 0 memory needed by the queue is given by the formula in the section of this manual called “Memory Usage.”
8. q_vcreate() creates a variable length message queue. Use q_create() to create an ordinary queue.

Multiprocessor Considerations

1. The Q_GLOBAL attribute should be set only if the queue must be made known to other processor nodes in a multiprocessor configuration. If set, the queue's name and qid are sent to the master node for entry in its Global Object Table.
2. If the Q_GLOBAL attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry mc_nglbobj then the queue is not created and ERR_OBJTFULL is returned.
3. If the maximum message length as specified by maxlen might require transmission of a packet larger than the KI can transmit, as specified in the Multiprocessor Configuration Table entry mc_kimaxbuf, then the queue is not created and ERR_KISIZE is returned.

Callable From

- Task

See Also

q_create, *q_vident*, *q_vdelete*

q_vdelete Deletes a variable-length message queue.

```
#include <psos.h>
unsigned long q_vdelete(
   unsigned long qid    /* queue identifier */
   )
```


Description

This system call deletes a variable length message queue. Otherwise, it is identical to `q_delete()`.

Arguments

`qid`	Specifies the queue ID of the queue to delete.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Message storage is returned to region 0. Hence the calling task can be preempted by a high priority task waiting for memory.
2. The calling task can also be preempted after this call, if a task waiting at the deleted queue has higher priority.
3. Any pending messages are lost.
4. `q_vdelete()` deletes a variable length message queue. Use `q_delete()` to delete an ordinary queue.

Multiprocessor Considerations

If `qid` identifies a global queue, `q_vdelete()` will notify the master node so that the queue can be removed from its Global Object Table. Thus, deletion of a global queue always causes activity on the master node.

Callable From

- Task

See Also

q_delete, *q_vcreate*

q_vident

Obtains the queue ID of a variable-length message queue.

```
#include <psos.h>
unsigned long q_vident(
   char name[4],          /* queue name */
   unsigned long node,    /* node number */
   unsigned long *qid     /* queue identifier */
   )
```


Description

The intended purpose of this system call is to allow the calling task to obtain the queue ID of a variable length message queue. However, since a variable length message queue is just a special type of message queue, `q_ident()` and `q_vident()` are functionally identical. Both return the queue ID of the first variable length or fixed length queue encountered with the specified name.

Arguments

`name`	Specifies the user-assigned name of the message queue.
`node`	For multiprocessor systems, is a search order specifier. See *Multiprocessor Considerations*. In a single node system, this argument must be 0.
`qid`	Points to the variable where `q_vident()` stores the queue ID of the named queue.

Return Value

The system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a queue name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the queue name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate queue names. If duplicate names exist, a `q_vident()` call can return the `qid` of any queue with the duplicate name.

Multiprocessor Considerations

1. `q_vident()` converts a queue's name to its qid using a search order determined by the `node` input parameter as described in *pSOSystem System Concepts*. Because queues created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, the local kernel makes an `q_vident()` RSC to the master node.

Callable From

- Task

See Also

q_ident, *q_vcreate*

q_vinfo

Queries a Variable Length Message Queue Object.

```
#include <psos.h>
unsigned long q_vinfo(
   unsigned long vqid,              /* Message Queue ID */
   struct qvinfo *buf,              /* Object Information buffer */
   )
```


Description

This system call returns a Variable Length Message Queue object's information. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

vqid	Specifies the ID of the Variable Length Message Queue object being queried.
buf	Contains the Variable Length Message Queue information.

struct qvinfo has the following members:

```
struct qinfo   fqinfo;   /* Generic queue information */
unsigned long  maxlen    /* max. message length allowed */
```

fqinfo which is of type *struct qinfo* has the following members:

```
char           name[4];  /* Name of the queue */
unsigned long  flags;    /* Queue attributes */
unsigned long  wqlen;    /* No. of waiting tasks */
unsigned long  wtid;     /* ID of the first waiting task */
unsigned long  mqlen;    /* No. of messages in the queue */
unsigned long  mqmax;    /* Max no. of messages allowed */
unsigned long  tid_ntfy; /* Task to be notified of message arrival */
unsigned long  ev_ntfy;  /* Events to post to notify mesg arrival */
```

name specifies the name of the queue.

The field **flags** returns the options with which the queue was created. The field **wqlen** gives the number of tasks waiting on this queue to receive messages. **wtid** is the ID of the first waiting task. **wtid** is zero when there are no waiting tasks.

The value in **mqlen** is the number of messages in the queue to be picked up.The value in **mqmax** is the max. number of messages allowed for this queue. The number of free message buffers available anytime is given by **mqmax - mqlen**.

Whenever a message is posted onto the queue, the task denoted by **tid_ntfy**, is notified by the sending of an event **ev_ntfy**. A 0 value for **tid_ntfy** denotes that there is no task to notify on message arrival; and a 0 value for **ev_ntfy** denotes that there is no event to post.

maxlen gives the maximum length of the message that is allowed.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, `SC_PSOS_QUERY`, to `YES` in the file `<sys_conf.h>` to access the pSOS+ `q_vinfo()` service call.

Multiprocessor Considerations

`vq_info()` can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

`q_vcreate`, *`q_info`*, *`q_vsend`*

q_vnotify

Registers the events for notification of availability of a message.

```
#include <psos.h>
unsigned long q_vnotify(
    unsigned long qid,              /* ID of target queue */
    unsigned long tid,              /* ID of task to be notified */
    unsigned long events            /* bit-encoded events */
    )
```


Description

This system call registers a set of bit-encoded events and a task ID for the given variable length message queue. Once so registered, a notification is sent to the task whenever a message is posted to the queue, via an implicit call of the form: `ev_send(tid, events)`. The task can choose to receive the notification via `ev_receive()` call.

Arguments

`qid`	Specifies the ID of the variable length message queue for which the notification of availability of message is to be registered.
`tid`	Specifies the ID of the task that must be notified of the availability of the message. A value of 0 registers the calling task as the one to be notified.
`events`	Contains a list of bit-encoded events. A value of 0 turns off the event notification mechanism.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A message can be posted to a message queue, and hence the notification event can be generated only as a result of `q_vsend()` or `q_vurgent()` calls. Note

that the q_vbroadcast() call never posts a message to a queue; it only delivers the message to any tasks waiting to receive the message.

2. When event notification for availability of a message is requested, the notification is generated only when there are no tasks waiting to receive a message. If there are tasks waiting to receive a message, one such task is awakened by pSOS+ and given the message but no notification is generated.

3. The ID of the task specified by tid argument is not verified during q_vnotify() call, but it is verified every time a notification is sent.

4. A notification does not guarantee that the message will remain available to be grabbed by the task that was notified. A higher priority task, or even an ISR, can potentially obtain the message before the notified task has a chance to run.

5. This call can be used to register a notification for an ordinary message queue, since variable length message queues are a special type of message queue. However, for the sake of consistency, a separate call named q_notify() has been provided that has the same functionality as q_vnotify().

Multiprocessor Considerations

None, q_vnotify() can be called only from the local node.

Callable From

- Task

See Also

q_vsend, *q_vurgent*, *q_notify*, *sm_notify*, *as_notify*, *ev_send*, *ev_receive*

q_vreceive

Requests a message from a variable-length message queue.

```
#include <psos.h>
unsigned long q_vreceive(
   unsigned long qid,     /* queue identifier */
   unsigned long flags,   /* queue attributes */
   unsigned long timeout, /* timeout in clock ticks */
   void *msg_buf,         /* message buffer */
   unsigned long buf_len, /* length of buffer */
   unsigned long *msg_len /* length of message */
   )
```


Description

This system call enables a task or an ISR to obtain a message from a *variable length* message queue. Otherwise, it is identical to `q_receive()`.

Arguments

`qid` Specifies the queue ID of the target queue.

`flags` Specifies whether `q_vreceive()` will block waiting for a message. `flags` should have one of the following values (defined in `<psos.h>`):

`Q_NOWAIT`	Don't wait for message.
`Q_WAIT`	Wait for message.

`timeout` Specifies the timeout interval, in units of clock ticks. If `timeout` is zero and `Q_WAIT` is selected, then `q_vreceive()` will wait forever. `timeout` will be ignored if `Q_NOWAIT` is selected.

`msg_buf` Points to the buffer that receives the message.

`buf_len` Specifies the length of the buffer `msg_buf` points to. If `buf_len` is less than the length of the message queued at the queue head, `ERR_BUFSIZ` is returned to the caller.

`msg_len` Points to the variable where `q_receive()` stores the actual length of the received message.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If it is necessary to block the calling task, `q_vreceive()` will enter the calling task in the queue's task-wait queue. If the queue was created with the `Q_FIFO` attribute, then the caller is simply entered at the tail of the wait queue. If the queue was created with the `Q_PRIOR` attribute, then the task will be inserted into the wait queue by priority.
2. The pSOS+ kernel must copy the message from the queue into the caller's buffer. Longer messages take longer to copy. User's should account for the copy time in their design, especially when calling from an ISR.
3. `q_vreceive()` requests a message from a variable length message queue. Use `q_receive()` to request a message from an ordinary queue.

Multiprocessor Considerations

If `qid` identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to request a message from that queue. If the `Q_WAIT` attribute is elected, then the pSOS+m kernel on the target node must use an agent to wait for the message. If that node is temporarily out of agents, the call will fail. The number of agents on each node is defined by the `mc_nagent` entry in the Multiprocessor Configuration Table.

Callable From

- Task.
- ISR, if `Q_NOWAIT` is set.
- KI, if `Q_NOWAIT` is set.
- Callout, if `Q_NOWAIT` is set.

See Also

`q_receive`, *`q_vsend`*, *`q_vnotify`*

q_vsend

Posts a message to a specified variable-length message queue.

```
#include <psos.h>
unsigned long q_vsend(
   unsigned long qid,     /* queue identifier */
   void *msg_buf,         /* message buffer */
   unsigned long msg_len  /* length of message */
   )
```


Description

This system call is used to send a message to a specified variable length message queue. Other than the queue type, `q_vsend()` operates just like `q_send()`.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Points to the message to send.
`msg_len`	Specifies the length of the message. It must not exceed the queue's maximum message length.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the caller is a task, it may be preempted as a result of this call.
2. The pSOS+ kernel must copy the message into a queue buffer or the receiving task's buffer. Longer messages take longer to copy. User's should account for the copy time in their design, especially when calling from an ISR.
3. `q_vsend()` sends a message to a variable length message queue. Use `q_send()` to send a message to an ordinary queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to post the input message to that queue.
2. If a task awakened by this call does not reside on the local node, the local kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_vsend()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task.
- ISR, if the target queue is local to the node from which the `q_vsend()` call is made.
- KI, if the target queue is local to the node from which the `q_vsend()` call is made.
- Callout, if the target queue is local to the node from which the `q_vsend()` call is made.

See Also

q_send, *q_vreceive*, *q_vnotify*

q_vurgent

Posts a message at the head of a variable-length message queue.

```
#include <psos.h>
unsigned long q_vurgent(
   unsigned long qid,     /* queue identifier */
   void *msg_buf,         /* message buffer */
   unsigned long msg_len  /* length of message */
   )
```


Description

This system call is identical in all respects to `q_vsend()` with one and only one exception: if one or more messages are already posted at the target queue, then the new message will be inserted into the queue's message queue in front of all such queued messages.

Arguments

`qid`	Specifies the queue ID of the target queue.
`msg_buf`	Points to the message to send.
`msg_len`	Specifies the length of the message. It must not exceed the queue's maximum message length.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `q_vurgent()` is useful when the message represents an urgent errand and must be serviced ahead of the normally FIFO ordered messages.
2. If the caller is a task, it may be preempted as a result of this call.

3. The pSOS+ kernel must copy the message into a queue buffer or the receiving task's buffer. Longer messages take longer to copy. User's should account for the copy time in their design, especially when calling from an ISR.

4. `q_vurgent()` sends an urgent message to a variable length message queue. Use `q_urgent()` to send an urgent message to an ordinary queue.

Multiprocessor Considerations

1. If `qid` identifies a global queue residing on another processor node, the local kernel will internally make an RSC to that remote node to post the input message to that queue.

2. If a task awakened by this call does not reside on the local node, the local kernel will internally pass the message to the task's node of residence, whose pSOS+m kernel will ready the task and give it the relayed message. Thus, a `q_vurgent()` call, whether it is on the local or a remote queue, may cause pSOS+m activities on another processor node.

Callable From

- Task.
- ISR, if the target queue is local to the node from which the `q_vurgent()` call is made.
- KI, if the target queue is local to the node from which the `q_vurgent()` call is made.
- Callout, if the target queue is local to the node from which the `q_vurgent()` call is made.

See Also

`q_urgent`, *`q_vreceive`*, *`q_vsend`*, *`q_vnotify`*

rn_create

Creates a memory region.

```
#include <psos.h>
unsigned long rn_create(
   char name[4],            /* region name */
   void *saddr,             /* starting address */
   unsigned long length,    /* region's size in bytes */
   unsigned long unit_size, /* region's unit of allocation */
   unsigned long flags,     /* region attributes */
   unsigned long *rnid,     /* region ID */
   unsigned long *asiz      /* allocatable size */
   )
```


Description

This service call enables a task to create a memory region, from which variable-sized memory segments can be allocated for use by the application. The pSOS+ kernel takes a portion from the beginning of this region to use as its Region Control Block (RNCB.) All relevant region arguments such as unit size and whether tasks will wait by FIFO or task priority order are established using this call.

Arguments

`name` Specifies the user-assigned name of the new region.

`saddr` Specifies the starting address of the region's memory area. `saddr` must be on a long word boundary.

`length` Specifies the region's size, in bytes.

`unit_size` Specifies the region's unit of allocation in bytes. `unit_size` must be a power of 2 and greater than or equal to 16. All allocation will be in multiples of `unit_size`.

`flags` Specifies the region's attributes. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify queuing by task priority, you place `RN_PRIOR` in `flags`. To specify queuing by task priority and to enable deletion of the region even if segments are allocated, you place both `RN_PRIOR` and `RN_DEL` in `flags`, using the following syntax:

```
RN_PRIOR | RN_DEL
```

	RN_PRIOR	Tasks are queued by priority.
	RN_FIFO	Tasks are queued by FIFO order.
	RN_DEL	Region can be deleted with segments outstanding.
	RN_NODEL	Region cannot be deleted with segments outstanding.
rnid		Points to the variable where rn_create() stores the region ID of the named region.
asiz		Points to the variable where rn_create() stores the actual number of allocatable bytes available in the region.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a region name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the region name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate region names. If duplicate names exist, an rn_ident() call can return the rnid of any region with the duplicate name.
3. A region must consist of physically contiguous memory locations.
4. The maximum input length for a region is 32767 times the region's unit size. A length that exceeds this limit for a given unit size is treated as an error, the remedy for which is a bigger unit size.
5. When the RN_DEL attribute is specified, a region can be deleted while segments are outstanding; otherwise, the pSOS+ kernel requires all segments to be returned before the region can be deleted.

Multiprocessor Considerations

Regions are strictly local resources, and cannot be exported. Therefore, any allocation calls must come only from the local node. However, if a region's memory is

reachable from other nodes, then any segments allocated from it can be passed between nodes for direct access explicitly by the user's code.

Callable From

- Task

See Also

rn_ident, *rn_getseg*

rn_delete Deletes a memory region.

```
#include <psos.h>
unsigned long rn_delete (
   unsigned long rnid   /* region ID */
   )
```

Description

This system call deletes the memory region with the specified region ID. Unless the region was created with the `RN_DEL` attribute set, `rn_delete()` is rejected if any segments allocated from the region have not been returned. The calling task does not have to be the creator of the region to be deleted.

Arguments

`rnid` Specifies the region ID of the region to be deleted.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Once created, a region generally is used by multiple tasks for storing or passing data. Reasons for deleting a region are rare. Deleting a region is dangerous except as part of a partial or full system restart.
2. The special region with `rnid` equal to 0 cannot be deleted.

Multiprocessor Considerations

None, since regions are local resources. `rn_delete()` can be called only from the local node.

Callable From

- Task

See Also

rn_create

rn_getseg

Allocates a memory segment to the calling task.

```
#include <psos.h>
unsigned long rn_getseg(
   unsigned long rnid,    /* region identifier */
   unsigned long size,    /* requested size, in bytes */
   unsigned long flags,   /* segment attributes */
   unsigned long timeout, /* timeout in clock ticks */
   void **seg_addr        /* allocated segment address */
   )
```

Description

This system call allocates a memory segment of the specified size from the specified memory region. An allocated segment's size is always the nearest multiple of the region's unit size, which is an input argument to the `rn_create()` call.

If the calling task selects the `RN_NOWAIT` attribute, then `rn_getseg()` returns unconditionally (whether or not allocation was successful). If the calling task elects the `RN_WAIT` attribute, and a subsequent request cannot be satisfied, the task is blocked until either a segment is allocated, or a timeout occurs (if the timeout attribute is elected).

Arguments

`rnid` Specifies the region ID from which the memory segment is allocated.

`size` Specifies the segment size, in bytes.

`flags` Specifies the segment's attributes. The `flags` argument must assume one of the following values, defined in `<psos.h>`.

- `RN_NOWAIT` Don't wait for a segment.
- `RN_WAIT` Wait for a segment.

`timeout` Specifies the timeout, in units of clock ticks. If `timeout` is 0 and `flags` is set to `RN_WAIT`, then `rn_getseg()` will wait forever. The `timeout` argument is ignored if `RN_NOWAIT` is used.

`seg_addr` Points to the variable where `rn_getseg()` stores the starting address of the memory segment.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. See *pSOSystem System Concepts* for a description of the allocation algorithm for regions.
2. An allocated segment's size will always be a multiple of the region's unit size. It can, therefore, be greater than `size`.
3. An allocated segment always starts on a long word boundary.
4. If the calling task must wait, it will either wait by FIFO or priority order, depending on the attribute elected when the region was created.

Multiprocessor Considerations

Regions are strictly local resources, and cannot be exported. Therefore, any allocation calls must come only from the local node. However, if a region's memory is reachable from other nodes, then any segments allocated from it can be passed between nodes for direct access explicitly by the user's code.

Callable From

- Task

See Also

rn_create, *rn_retseg*

rn_ident

Obtains the region identifier of a named region.

```
#include <psos.h>
unsigned long rn_ident(
   char name[4],          /* region name */
   unsigned long *rnid    /* region identifier */
   )
```

Description

This system call enables the calling task to obtain the region ID of a memory region for which the caller has only the region name. This region ID can then be used in all other operations relating to the memory region.

Most system calls, except `rn_create()` and `rn_ident()`, reference a region by its region ID. `rn_create()` returns the region ID to a region's creator. For other tasks, one way to obtain the region ID is to use `rn_ident()`.

Arguments

`name`	Specifies the user-assigned name of the region.
`rnid`	Points to the variable where `rn_ident()` stores the region ID of the named region.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a region name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the region name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate region names. If duplicate names exist, an `rn_ident()` call can return the rnid of any region with the duplicate name.
3. The region with `rnid` equal 0 is special. This region is statically specified in the pSOS+ Configuration Table, and is used for pSOS+ data structures and task stacks.

Multiprocessor Considerations

None, since regions are strictly local resources. Only the local object table is searched.

Callable From

- Task

See Also

rn_create

rn_info Queries a Region Object.

```
#include <psos.h>

unsigned long rn_info(
    unsigned long rnid,             /* Region ID */
    struct rninfo *buf,             /* Object Information buffer */
    )
```

Description

This system call returns information of a specified region. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

rnid	Specifies the ID of the Region object being queried.
buf	Contains the Region information.

struct rninfo has the following members.

```
char            name[4];        /* Name of the region */
unsigned long   flags;          /* Region attributes */
unsigned long   wqlen;          /* No. of waiting tasks */
unsigned long   wtid;           /* ID of the first waiting task */
void            *start_addr;    /* Start address */
unsigned long   unit_size;      /* Unit size in bytes */
unsigned long   total_units;    /* Total units */
unsigned long   free_bytes;     /* Free bytes */
unsigned long   largest;        /* Size of largest chunk in bytes */
unsigned long   length;         /* Total length of region in bytes */
```

name has the region name. The attributes with which the region was created are in **flags**. **wqlen** is the number of tasks waiting on this region to get a segment. **wtid** stores the ID of the first waiting task. **wtid** is zero when there are no waiting tasks.

saddr is the starting address of the region and **total_units** is the total number of units in this region respectively. **unit_size** is the size of each unit in bytes. The number of free bytes in this region is returned in **free_bytes**. **largest** is the size of largest cluster in bytes. **length** is the total length of the region.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, SC_PSOS_QUERY, to YES in the file <sys_conf.h> to access the pSOS+ m_info() service call.

Multiprocessor Considerations

rn_info() can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

rn_create

rn_retseg

Returns a memory segment to the region from which it was allocated.

```
#include <psos.h>
unsigned long rn_retseg(
   unsigned long rnid, /* region identifier */
   void *seg_addr      /* segment address */
   )
```

Description

This system call returns a memory segment back to the region from which it was allocated. The pSOS+ Region Manager then performs whatever compaction is possible, and puts the resulting free memory block in the region's free list for future allocation.

The segment address specified must be identical to the one returned by the original `rn_getseg()` call. Otherwise, the pSOS+ kernel will reject the segment.

Arguments

`rnid`	Specifies the segment's region of origin.
`seg_addr`	Specifies the starting address of the segment, as returned by `rn_getseg()`.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Refer to *pSOSystem System Concepts* for the algorithms used to merge neighboring free segments.
2. There is no notion of segment ownership. A segment can be returned by a task other than the one that originally allocated it.

3. If there are tasks waiting for memory from this region, then such requests will be re-examined and allocation granted where possible — in the order of the wait queue (FIFO or by task priority).

4. Note that the calling task may be preempted if a task waiting for memory segment is unblocked as a result of the returned segment, and that task has higher priority.

Multiprocessor Considerations

None, since regions are strictly local resources. `rn_retseg()` can be called only from the local node.

Callable From

- Task

See Also

`rn_getseg`

sm_av (pSOS+m kernel only) Asynchronously releases a semaphore token.

```
#include <psos.h>
unsigned long sm_av(
   unsigned long smid    /* semaphore identifier */
   )
```

Description

This system call is used to asynchronously release a semaphore token. It is identical to `sm_v()` except the call is made asynchronously. Refer to the description of `sm_v()` for further information. This call is only supported by the pSOS+m kernel (the multiprocessor version).

Arguments

`smid` Specifies the semaphore ID of the semaphore token to release.

Return Value

When called in a system running the pSOS+m kernel, this call always returns 0. The pSOS+ kernel (the single processor version) returns ERR_SSFN.

Error Codes

Should the call fail, if present, the node's MC_ASYNCERR routine is invoked and error codes are reported. Refer to Appendix A for the error codes.

If the MC_ASYNCERR routine is not provided, the pSOS+ m kernel generates a fatal error.

Notes

The calling task can be preempted as a result of this call.

Multiprocessor Considerations

1. If `smid` identifies a global semaphore residing on another processor node, the pSOS+ kernel internally makes an RSC to that remote node to release the semaphore.
2. If the task awakened by this call does not reside on the local node, then the pSOS+m kernel internally alerts the task's node of residence, whose pSOS+m kernel will ready the task and give it the acquired semaphore token. Thus, an `sm_v()` call, whether it is to either the local or a remote semaphore, may cause pSOS+m activities on another processor node.

Callable From

- Task

See Also

`sm_v`, *`sm_p`*, *`sm_notify`*

sm_create Creates a semaphore.

```
#include <psos.h>
unsigned long sm_create(
   char name[4],          /* semaphore name */
   unsigned long count,   /* number of tokens */
   unsigned long flags,   /* semaphore attributes */
   unsigned long *smid    /* semaphore identifier */
   )
```

Description

This system call creates a semaphore by allocating and initializing a Semaphore Control Block (SMCB) according to the specifications supplied with the call.

Like all objects, a semaphore has a user-assigned name, and a pSOS+-assigned semaphore ID returned by `sm_create()`. Several flag bits specify the characteristics of the semaphore, including whether tasks will wait for the semaphore by task priority or strictly FIFO.

Arguments

`name` Specifies the user-assigned name of the new semaphore.

`count` Specifies the initial semaphore token count.

`flags` Specifies the semaphore's attributes. `flags` is formed by OR-ing the following symbolic constants (one from each pair), defined in `<psos.h>`:

`SM_GLOBAL`	Semaphore can be addressed by other nodes.
`SM_LOCAL`	Semaphore can be addressed by local node only.
`SM_PRIOR`	Tasks are queued by priority.
`SM_FIFO`	Tasks are queued by FIFO.
`SM_UNBOUNDED`	Semaphore count is not bounded.
`SM_BOUNDED`	Semaphore count is bounded. When the `SM_BOUNDED` flag is specified the `count` parameter specifies the initial semaphore token count, as well as the upper bound of available tokens.

`smid` Points to the variable where `sm_create()` stores the semaphore ID of the named semaphore.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a semaphore name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the semaphore name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate semaphore names. If duplicate names exist, an `sm_ident()` call can return the `smid` of any semaphore with the duplicate name.
3. The maximum number of semaphores that can be simultaneously active is defined by the `kc_nsema4` entry in the pSOS+ Configuration Table.
4. The `count` argument is unsigned, and thus can only be 0 or a positive value.
5. The `SM_GLOBAL` attribute is ignored by the single-processor version of the pSOS+ kernel.
6. A semaphore created with `SM_BOUNDED` flag set, and the `count` parameter set to one behaves like a binary semaphore.

Multiprocessor Considerations

1. The `SM_GLOBAL` attribute should be set only if the semaphore must be made known to other processor nodes in a multiprocessor configuration. If set, the semaphore's name and `smid` are sent to the master node for entry in its Global Object Table.
2. If the `SM_GLOBAL` attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry `mc_nglbobj` then the semaphore is not created and `ERR_OBJTFULL` is returned.

Callable From

- Task

See Also

sm_delete, *sm_ident*

sm_delete Deletes a semaphore.

```
#include <psos.h>
unsigned long sm_delete(
   unsigned long smid   /* semaphore ID */
   )
```


Description

This system call deletes the semaphore with the specified semaphore ID, and frees the SMCB to be reused. `sm_delete()` takes care of cleaning up the semaphore. If there are tasks waiting, they will be unblocked and given an error code.

The calling task does not have to be the creator (parent) of the semaphore to be deleted. However, a semaphore must be deleted from the node on which it was created.

Arguments

`smid` Specifies the semaphore ID of the semaphore to be deleted.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Once created, a semaphore is generally used by multiple tasks for communication and synchronization. There is rarely a reason for deleting a semaphore, even when it is no longer used, except to allow reuse of the SMCB.
2. The calling task can be preempted, if a task waiting at the deleted semaphore has higher priority.

Multiprocessor Considerations

If `smid` identifies a global semaphore, `sm_delete` will notify the master node so that the semaphore can be removed from its Global Object Table. Thus, deletion of a global semaphore always causes activity on the master node.

Callable From

- Task

See Also

`sm_create`

sm_ident

Obtains the semaphore identifier of a named semaphore.

```
#include <psos.h>
unsigned long sm_ident(
   char name[4],          /* semaphore name */
   unsigned long node,    /* node selector */
   unsigned long *smid    /* semaphore ID */
   )
```


Description

This system call enables the calling task to obtain the semaphore ID of a semaphore it only knows by name. The semaphore ID can then be used in all other operations relating to this semaphore.

Most system calls, except `sm_create()` and `sm_ident()`, reference a semaphore by the semaphore ID. `sm_create()` returns the semaphore ID to the semaphore creator. For other tasks, one way to obtain the semaphore ID is to use `sm_ident()`.

Arguments

`name`	Specifies the user-assigned name of the semaphore.
`node`	In multiprocessor systems, is a search order specifier. See *Multiprocessor Considerations*. In a single node system, this argument must be 0.
`smid`	Points to the variable where `sm_ident()` stores the semaphore ID of the named semaphore.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a semaphore name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the semaphore name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate semaphore names. If duplicate semaphore names exist, an `sm_ident()` call can return the `smid` of any semaphore with the duplicate name.

Multiprocessor Considerations

1. `sm_ident()` converts a semaphore's name to its smid by using a search order determined by the `node` input parameter, as described in *pSOSystem System Concepts*. Because semaphores created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, then the pSOS+m kernel makes a `sm_ident()` RSC to the master node.

Callable From

- Task

See Also

sm_create

sm_info Queries a Semaphore Object.

```
#include <psos.h>
unsigned long sm_info(
   unsigned long smid,              /* Semaphore ID */
   struct sminfo *buf,              /* Object Information buffer */
   )
```


Description

This system call returns information of a specified Semaphore object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

smid	Specifies the ID of the Semaphore object being queried.
buf	Contains the Semaphore information.

struct sminfo has the following members.

```
char             name[4];     /* Name of semaphore */
unsigned long    flags;       /* Semaphore attributes */
unsigned long    wqlen;       /* No. of waiting tasks */
unsigned long    wtid;        /* ID of the first waiting task */
unsigned long    count;       /* Semaphore count */
unsigned long    maxcount;    /* Limit for bounded semaphores */
unsigned long    tid_ntfy;    /* Task to notify of availability */
unsigned long    ev_ntfy;     /* Ev to post to notify availability */
```

name returns the name of the semaphore. **flags** specify the attributes with which the semaphore was created. **wqlen** is the total number of tasks waiting to acquire this semaphore. **wtid** stores the ID of the first waiting task. **wtid** is zero when there are no waiting tasks. If the pSOS® operating system is not a 2.5 or a later release, **wtid** returns 0 when the waiting task is an agent task (in the case of pSOS+m), otherwise **wtid** represents the remote task ID.

count is the current count of the semaphore. **maxcount** is the maximum count allowed for bounded semaphores. This value is meaningless for unbounded semaphores.

Whenever a semaphore token becomes available because of the sm_v() operation, the task denoted by **tid_ntfy**, is notified by the sending of an event **ev_ntfy**. A 0 value for **tid_ntfy** denotes that there is no task to notify on semaphore availability; and a 0 value for **ev_ntfy** denotes that there is no event to send.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, SC_PSOS_QUERY, to YES in the file <sys_conf.h> to access the pSOS+ sm_info() service call.

Multiprocessor Considerations

sm_info() can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

sm_create, *sm_v*

sm_notify

Registers the events for notification of semaphore availability.

```
#include <psos.h>
unsigned long sm_notify(
   unsigned long smid,              /* ID of target semaphore */
   unsigned long tid,               /* ID of task to be notified */
   unsigned long events             /* bit-encoded events */
   )
```


Description

This system call registers a set of bit-encoded events and a task ID for the given semaphore. Once so registered, a notification is sent to the task whenever a semaphore token becomes available, via an implicit call of the form: `ev_send(tid, events)`. The task can choose to receive the notification via `ev_receive()` call.

Arguments

`smid`	Specifies the ID of the sempahore for which the notification of semaphore-token availability is to be registered.
`tid`	Specifies the ID of the task that must be notified of the availability of the semaphore token. A value of 0 registers the calling task as the one to be notified.
`events`	Contains a list of bit-encoded events. A value of 0 turns off the event notification mechanism.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A semaphore token can become available, and hence the notification event can be generated only as a result of `sm_v()` call.
2. When event notification for availability of a semaphore token is requested, the notification is generated only when there are no tasks waiting to obtain the sempahore token. If there are tasks waiting to obtain semaphore token, one such task is awakened by pSOS+ and given the semaphore token but no notification is generated.
3. The ID of the task specified by tid argument is not verified during `sm_notify()` call, but it is verified every time a notification is sent.
4. A notification does not guarantee that the semaphore token will remain available to be grabbed by the task that was notified. A higher priority task, or even an ISR, can potentially obtain the semaphore token before the notified task has a chance to run.

Multiprocessor Considerations

None, `sm_notify()` can be called only from the local node.

Callable From

- Task

See Also

`sm_v`, *`ev_send`*, *`ev_receive`*

sm_p Acquires a semaphore token.

```
#include <psos.h>
unsigned long sm_p(
   unsigned long smid,   /* semaphore identifier */
   unsigned long flags,  /* attributes */
   unsigned long timeout /* timeout */
   )
```


Description

This system call enables a task or an ISR to acquire a semaphore token. A calling task can specify whether or not it wants to wait for the token. If the semaphore token count is positive, then this call returns the semaphore token immediately. If the semaphore token count is zero and the calling task specified SM_NOWAIT, then sm_p() returns with an error code. If SM_WAIT is elected, the task will be blocked until a semaphore token is released, or if the timeout argument is specified, until timeout occurs, whichever occurs first.

Arguments

smid	Specifies the semaphore ID of the semaphore.	
flags	Specifies whether sm_p() will block waiting for a token. flags should have one of the following values (defined in <psos.h>):	
	SM_WAIT	Block until semaphore is available.
	SM_NOWAIT	Return with error code if semaphore is unavailable.
timeout	Specifies the timeout interval, in units of clock ticks. If timeout is zero and flags is set to SM_WAIT, then sm_p() will wait forever. timeout will be ignored if flags is set to SM_NOWAIT.	

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

If it is necessary to block the calling task, `sm_p()` will enter the calling task in the semaphore's task-wait queue. If the semaphore was created with the `SM_FIFO` attribute, then the task is simply entered at the tail of the wait queue. If the semaphore was created with the `SM_PRIOR` attribute, then the task is inserted into the wait queue by priority.

Multiprocessor Considerations

If `smid` identifies a global semaphore residing on another processor node, the local kernel will internally make an RSC to that remote node to acquire a semaphore token. If the `SM_WAIT` attribute is used, then the pSOS+ kernel on the target node must use an agent to wait for the semaphore token. If that node is temporarily out of agents, the call will fail. The number of agents on each node is defined by the `mc_nagent` entry in the node's Multiprocessor Configuration Table.

Callable From

- Task.
- ISR, if `SM_NOWAIT` is set and the semaphore is local to the node from which the `sm_p()` call is made.
- KI, if `SM_NOWAIT` is set and the semaphore is local to the node from which the `sm_p()` call is made.
- Callout, if `SM_NOWAIT` is set and the semaphore is local to the node from which the `sm_p()` call is made.

See Also

`sm_v`

sm_v Releases a semaphore token.

```
#include <psos.h>
unsigned long sm_v(
   unsigned long smid    /* semaphore identifier */
   )
```


Description

This system call is used to release a semaphore token. If a task is already waiting at the semaphore, it is unblocked and made ready to run. If there is no task waiting, then the semaphore token count is simply incremented by 1.

If, as a result of this call, a semaphore token becomes available, and a set of task and events had been registered via a prior call to `sm_vnotify()` to notify the availability of the semaphore token, pSOS+ also performs an `ev_send()` operation to send the registered events to the registered task.

Arguments

`smid`	Specifies the semaphore ID of the semaphore token to release.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the caller is a task, it may be preempted as a result of this call.
2. If the semaphore was created with `SM_BOUNDED` flag and there is no task waiting, one of the following actions may be taken:
   - if the semaphore token count is less than the bound of available tokens, then the semaphore token count is simply incremented by 1
   - if the semaphore token count is equal to the bound of available tokens, an error is returned

3. If the semaphore was created with SM_UNBOUNDED flag and there is no task waiting, then the semaphore token count is simply incremented by 1.

Multiprocessor Considerations

1. If smid identifies a global semaphore residing on another processor node, then the pSOS+ kernel will internally make an RSC to that remote node to release the semaphore.

2. If the task awakened by this call does not reside on the local node, the local kernel will internally alert the task's node of residence, whose pSOS+ kernel will ready the task and give it the acquired semaphore token. Thus, an sm_v() call, whether it is to a local or remote semaphore, may cause pSOS+ activities on another node.

Callable From

- Task.
- ISR, if semaphore is local to the node from which the sm_v() call is made.
- KI, if the semaphore is local to the node from which the sm_v() call is made.
- Callout, if the semaphore is local to the node from which the sm_v() call is made.

See Also

sm_p, *sm_notify*

sys_info

Obtains pSOS+ system information.

```
#include <psos.h>
unsigned long sys_info(
   unsigned long key,              /* Information key */
   void *buffer,                   /* Information buffer */
   unsigned long buflen,           /* Length of buffer */
   unsigned long *actlen           /* Pointer to length returned */
   )
```


Description

This system call obtains the requested system information from the pSOS+ kernel, as specified by the **key** argument, and returns it in the user-supplied memory area, pointed to by **buffer**, whose length is specified by the **buflen** argument. Depending on the **key** argument, the information returned could be a null-terminated string, or an array of structures of a particular type, or a more complex record.

The pSOS+ version is returned as a null-terminated string, whereas the node roster, the run-time IO jump table, and the callout information, are returned as an array of structures of an appropriate type. If the buffer does not have enough space to hold all the information, only an integral number of structures are placed in the buffer.

For the Device Name Table, information is returned as an array of a fixed-length information structure of type **psos_dnt_entry. devname** holds the null-terminated device name string returned. The length of this string is, (kc_dnlen +1) rounded to the next closest long-word size multiple, where kc_dnlen denotes the value of the maximum device name length in the pSOS+ configuration table. If the buffer does not have enough space to hold all the information, only those many complete structures such that the total length of the buffer would not exceed **buflen**, are placed in the buffer.

In every case, the location pointed to by **actlen** is updated to contain the actual number of bytes of information that is returned in the buffer.

Arguments

`key` Specifies the key for the pSOS+ information requested. It can be one of the following symbolic constants defined in <psos.h>

`PSOS_VERSION`	Obtain the pSOS+ version number string. A null-terminated string is returned.

	PSOS_ROSTER	Obtain the pSOS+ node roster (pSOS+m only). An array of structures of type **psos_node_entry** is returned.
	PSOS_IOJT	Obtain the run-time pSOS+ IO jump table. An array of structures of type **psos_iojt_entry** is returned.
	PSOS_DNT	Obtain the run-time pSOS+ device name table. A collection of structures of equal length, which is the sum of the lengths of the structure **psos_dnt_entry** and the long-word-sized ceiling of the maximum device name length (which is a pSOS+ configuration parameter), is returned.
	PSOS_CALLOUT	Obtain the information on all registered pSOS+ callouts. An array of structures of type **psos_co_entry** is returned.
buffer	User-supplied buffer address, to store the roster information.	
buflen	Contains the length of the user-supplied buffer.	
actlen	Contains pointer to the location where the actual number of bytes of information stored in the buffer, is returned.	

Structures Returned

Structures Returned - PSOS_ROSTER

s*truct psos_node_entry* has the following members.

```
unsigned short     nodenum;      /* Node number */
unsigned short     seqnum;       /* Sequence number of node */
```

nodenum specifies the node number of a live node, and **seqnum** its current sequence number.

Structures Returned - PSOS_IOJT

struct psos_iojt_entry has the following members.

```
struct iojent      iojte;         /* IO jump table entry info */
unsigned long      devnum;        /* Device number of IOJT entry */
```

devnum specifies the device number associated with the pSOS+ IO jump table entry.

struct iojent has the following members.

```
void              (*dev_init);   /* Device init procedure */
void              (*dev_open);   /* Device open procedure */
void              (*dev_close);  /* Device close procedure */
void              (*dev_read);   /* Device read procedure */
void              (*dev_write);  /* Device write procedure */
void              (*dev_ioctl);  /* Device ioctl procedure */
unsigned long     rsvd1;         /* reserved, set to 0 */
unsigned short    rsvd2;         /* reserved, set to 0 */
unsigned short    flags;         /* IOJ table entry flags */
```

The first 6 function pointer members, point to the following device procedures: initialization, open, close, read, write, and ioctl. **flags** is specifies the entry properties, like device type, and whether this device was automatically initialized at either pSOS+ initialization or when the IO jump table entry was bound to the device.

Structures Returned - PSOS_DNT

struct psos_dnt_entry has the following members.

```
unsigned long     devnum;        /* Major-minor device number */
char              devname[ ];    /* Device name string */
```

devnum specifies the major-minor device number of associated with the Device Name Table entry. **devname** stores this device's null-terminated name string.

Structures Returned - PSOS_CALLOUT

struct psos_co_entry has the following members.

```
unsigned long  coid;           /* Unique ID of this callout */
unsigned long  type;           /* Callout type defined in psos.h */
void    (*func)(void *, CO_INFO *);/* Pointer to callout function */
void           *arg;           /* Argument to callout function */
```

coid is the ID of this callout**. type** is the type of the callout function, whose valid values are defined in **psos.h. func** is a pointer to the callout function. **arg** specifies the argument passed to the callout function.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, SC_PSOS_QUERY, to YES in the file <sys_conf.h> to access the pSOS+ sys_info() service call.

Multiprocessor Considerations

1. Version information is independent of multi-processor considerations.
2. Each node has a local copy of the node roster (node numbers of all live nodes on the system), however the slave nodes do not have the sequence number information. So the slave nodes will not return sequence number information about nodes other than self, when **key** has value, PSOS_ROSTER.
3. Whenever **key** has value, PSOS_IOJT, PSOS_DNT or PSOS_CALLOUT, the respective information resident on the local node is returned. sys_info() does not initiate any RSCs within the pSOS+ kernel.

Callable From

- Task
- Callout

See Also

ioj_bind, *ioj_lock*, *ioj_unlock*, *dnt_add*, *co_register*, *t_start*, *t_restart*, *t_delete*

t_addvar

Adds a new task variable to a task.

```
#include <psos.h>
unsigned long t_addvar(
   unsigned long tid,  /* task identifier */
   void **tv_addr      /* address of variable*/
   void *tv_value      /* initial value for task variable*/
   )
```


Description

This system call adds a task variable to a task. `t_addvar()` allocates to the specified task a storage area which is used to hold a private copy of the specified variable.

Arguments

`tid`	Specifies the ID of the task. If the tid equals 0, the task variable is added to the calling task itself.
`tv_addr`	Specifies the address of the variable.
`tv_value`	Specifies the value that the task variable is initialized with.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Since pSOS+ does not actually interpret the content of the variable. In addition to a pointer, the variable may be of any type which has the size and alignment of a pointer.
2. The task variable feature provides a convenient method for converting a global variable to a task-specific variable. pSOS+ saves the contents of a task variable when the task to which the variable is associated relinquishes control of the CPU. and restores its contents when that task regains control of the CPU.

3. The use of task variables can negatively impact the system's context switch timings. Therefore, the use of this feature should be carefully considered. The "Task-Specific-Data" object is a more run-time efficient alternative to task variables, and should be used when there are a large number of variables that need to be made task-specific.

Multiprocessor Considerations

None. An application may only add a task variable to a task on the local node.

Callable From

- Task

See Also

t_delvar, *tsd_create*

t_create

Creates a task.

```
#include <psos.h>
unsigned long t_create(
   char name[4],         /* task name */
   unsigned long prio,   /* task priority */
   unsigned long sstack, /* task supervisor stack size */
   unsigned long ustack, /* task user stack size */
   unsigned long flags,  /* task attributes */
   unsigned long *tid    /* task identifier */
   )
```


Description

This service call enables a task to create a new task. `t_create()` allocates to the new task a Task Control Block (TCB) and a memory segment for its stack(s). The task stack sizes and scheduling priority are established with this call. `t_create()` leaves the new task in a dormant state; the `t_start()` call must be used to place the task into the ready state for execution.

Arguments

`name` Specifies the user-assigned name of the task.

`prio` Specifies the task's initial priority within the range 1 - 230, with 230 the highest and 1 the lowest.

Priority level 0 is reserved for the pSOS+ daemon task IDLE. Priority levels 231 - 255 are reserved for a variety of high priority pSOSystem daemon tasks. While `t_create()` will allow creation of a task at these priorities, there should never be a need to do so.

`sstack` Specifies the task's supervisor stack size in bytes (see *Supervisor Stack Size* under *Target*.) `t_create()` internally calls `rn_getseg()` to allocate a segment from Region 0 to hold the task's stack and the user stack, if any.

`ustack` Specifies the task's user stack. `ustack` may be 0 if the task executes only in supervisor mode (see *Using sstack and ustack* under *Target*).

`flags` Specifies the task's attributes. `flags` is formed by OR-ing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that a task is global, you place the symbolic constant `T_GLOBAL` in `flags`. To specify that the task is global and uses the FPU processor, you place both `T_GLOBAL` and `T_FPU` in `flags`, using the following syntax:

```
T_GLOBAL | T_FPU
```

`T_GLOBAL`	Makes the task global: external tasks on other nodes can address it.
`T_LOCAL`	Restricts the task to the local node. The `T_GLOBAL` attribute is ignored by the single-processor kernel.
`T_FPU`	Informs the pSOS+ kernel that the task uses the FPU coprocessor (see *Using the T_FPU Flag* under *Target*.)
`T_NOFPU`	Informs the pSOS+ kernel that the task does not use the FPU coprocessor.

`tid` Points to the variable where `t_create()` stores the task ID assigned to the task.

Target

Using sstack and ustack

On most processors, a task can execute only in supervisor mode. Thus, a task can have only a supervisor stack. On these processors `ustack` is added to `sstack` to create a supervisor stack of the combined sizes. Exceptions to this usage are shown below.

68K On 68K processors, a task can execute in either user mode or supervisor mode. A user stack is not needed if the task never executes in the user mode, in which case `ustack` should be set to 0. If the task starts in the user mode, then `ustack` must be greater than 20 bytes. The supervisor/user mode is set in the `t_start()` system call.

CF PPC On ColdFire and PowerPC processors, a task can execute in either user mode or supervisor mode, but there are not separately defined stacks depending on this mode. `t_create()` simply adds `sstack` and `ustack` together and allocates a stack of that resultant size.

ARM

On ARM processors, a task can execute either in user mode or supervisor mode. A user stack is not needed if the task never executes in the user mode, in which case `ustack` is set to 0. If the task starts in the user mode, then `ustack` must be greater than 80 bytes to accommodate the requirements of the interrupt handler. The supervisor/user mode is set in the `t_start()` system call.

Supervisor Stack Size

Supervisor stack size is processor-dependent.

68K
CF

On 68K and ColdFire processors, the stack size should be no less than 128 bytes.

PPC
ARM
MIPS

On PowerPC, MIPS, and ARM processors, the stack size should be no less than 512 bytes.

960

On 960 processors, the stack size should be no less than 256 bytes.

x86

On x86 processors, the stack size should be no less than 128 bytes.

Using the T_FPU Flag

If the `T_FPU` flag is set, an additional save area is needed to save and restore FPU registers. It should be set if the task uses the FPU. When `t_create()` internally calls `rn_getseg()` to allocate a segment from Region 0 for stack allocation, the requested size of the segment is extended to accommodate the FPU save area. The size of the save area varies according to the processor being used.

68K

On 68K processors, the size of the FPU save area is 328 bytes.

On ColdFire processors, there is no support for FPU. Hence this feature is not supported in pSOS+.

On PowerPC processors, the size of the FPU save area is 264 bytes.

On ARM processors, the size of the FPU save area is 100 bytes.

On 960 processors, the size of the FPU save area is 48 bytes.

On 486 processors, and on 386 processors used with an 80387 FPU, the size of the FPU save area is 108 bytes.

On MIPS processors, the size of the FPU save area is 136 bytes.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a task name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the task name as a four-byte character array.
2. A null name (for example, 32-bit binary 0) should not be used, because it may be used elsewhere as an alias for the running task.
3. The pSOS+ kernel does not check for duplicate task names. If duplicate names exist, a `t_ident()` call can return the `tid` of any task with the duplicate name.

4. If you have installed any other components from Integrated Systems, the pSOS+ kernel needs to add an extension for each component for the task being created. When `t_create()` internally calls `rn_getseg()` to allocate a segment from Region 0 for stack allocation, the requested size of the segment is increased to accommodate the component extensions. These extension sizes can be determined from the user manuals for the respective components.

Multiprocessor Considerations

1. The `T_GLOBAL` attribute should be set only if the task must be made known to other processor nodes in a multiprocessor configuration. If set, the task's name and `tid` are sent to the master node for entry in its Global Object Table.

2. If the `T_GLOBAL` attribute is set and the number of global objects currently exported by the node equals the Multiprocessor Configuration Table entry `mc_nglbobj`, then the task is not created and `ERR_OBJTFULL` is returned.

Callable From

- Task

See Also

`t_start`, *`t_ident`*, *`rn_getseg`*

t_delete

Deletes a task.

```
#include <psos.h>
unsigned long t_delete(
   unsigned long tid    /* task identifier */
   )
```

Description

This service call enables a task to delete itself or another task. The pSOS+ kernel first dispatches any task deletion callouts, registered via the co_register() pSOS+ service, in the reverse order of registration. When all of the deletion callouts have finished, the pSOS+ kernel halts the task and reclaims its TCB, stack segment and any allocated timers.

The calling task does not have to be the creator (parent) of the task to be deleted. However, a task must be deleted from the node on which it was created.

Arguments

`tid` Specifies the task ID of the task to be deleted.

Return Value

This call returns 0 on success (unless the caller does a self-delete, in which case the call does not return) or returns an error on failure.

Error Codes

Refer to Appendix A.

Notes

1. If a task being deleted owns any mutexes, an error is returned and none of the mutexes are unlocked. The task needs to explicitly release all the mutexes before it could be safely deleted.
2. If the call is to delete self (suicide via `tid` equal to 0), there will be no return.
3. Task deletion should be carefully planned and considered. Indiscriminate use can lead to unpredictable results, especially when resources such as allocated

memory segments, buffers, or semaphores have not been correctly returned. If a task holds any resources from the pREPC+ library, the pHILE+ file system manager, or the pNA+ network manager, those resources must be returned before `t_delete()` is called. The commands that must be executed for pREPC+, pHILE+, and pNA+ resources are `fclose(0)`, `close_f(0)`, and `close(0)`, respectively. Following these commands, a `free()` command must be used to return pREPC+ memory. This order of execution is required because the pREPC+ library calls the pHILE+ file system manager, and the pHILE+ file system manager calls the pNA+ network manager (if NFS is in use.) If resources are not returned in the correct order, an error occurs. See Error Codes.

The following is an example of the correct sequence of calls for returning resources. This example applies to a case where all three components have allocated resources:

```
fclose(0);      /* return pREPC+ resources */
close_f(0);     /* return pHILE+ resources */
close(0);       /* return pNA+ resources */
free(-1);       /* return pREPC+ memory */
t_delete(0);     /* and execute self-deletion */
```

4. The task deletion callouts provide a way for tasks to perform any required cleanup that may be necessary for the orderly termination of the task, especially in the case where a `t_delete()` call is used to delete another task. The task deletion callouts execute in the context of the task being deleted and are free to use any system service calls, including the ones that are provided by other pSOSystem components.

5. A return from a `t_delete()` call that deleted another task may not signify that the task deletion has completed, if any task deletion callouts have been registered in the system. When any task deletion callouts have been registered, the task deletion occurs in three phases. In the first phase, all of the validation checks are made, the task is made ready to run (if it is not already), and any task deletion callouts are posted to run; the `t_delete()` call returns to the caller if any callouts were posted, or proceeds to phase three. The second phase involves execution of the task deletion callouts, and the relative priority of the task being deleted and the state of the other tasks in the system determines when the second phase is started. The completion of the second phase depends on the user-supplied tasks deletion callouts. After all of the task deletion callouts finish executing, the pSOS+ kernel regains control to complete the phase three of task deletion that involves halting the tasks and reclaiming the resources held by the task.

6. `t_delete()` calls the optional user-supplied callout procedure, whose address is defined in the `kc_deleteco` entry in the pSOS+ Configuration Table. This callout is different from the task deletion callouts registered via the co_register() service in many respects. This callout executes as an extension of the kernel code and is invoked after the kernel has removed the task from its object database and reclaimed its TCB. Since this callout executes as an extension to the kernel code, you can do only very limited processing in this routine. It is an obsolete feature that remains solely for the reason of maintaining backward compatibility with the previous versions of the pSOS+ kernel.

Multiprocessor Considerations

1. A task can be deleted only from the local node.
2. If `tid` identifies a global task, `t_delete` notifies the master node so that the task can be removed from its Global Object Table. Thus, deletion of a global task always causes activity on the master node.

Callable From

- Task

See Also

`t_restart`, *`mu_unlock`*

t_delvar

Deletes a task variable.

```
#include <psos.h>
unsigned long t_delvar(
   unsigned long tid,  /* task identifier */
   void **tv_addr      /* address of variable*/
   )
```


Description

This system call removes a task variable from a task. `t_delvar()` deallocates the private storage allocated for the task variable.

Arguments

`tid`	Specifies the ID of the task to which the task variable belongs. If the tid equals 0, the task variable of the calling task itself is removed.
`tv_addr`	Specifies the address of the variable.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only remove a task variable from a task on the local node.

Callable From

- Task

See Also

t_addvar

t_getreg

Gets a task's notepad register.

```
#include <psos.h>
unsigned long t_getreg(
   unsigned long tid,          /* task identifier */
   unsigned long regnum,       /* register number */
   unsigned long *reg_value    /* register contents */
   )
```


Description

This system call enables the caller to obtain the contents of a task's notepad register. Each task has 16 such software registers, held in the task's TCB. The purpose of these registers is to furnish every task with a set of named, permanent variables. Eight of these registers are reserved for system use. Eight are free to be used for application specific purposes.

Arguments

`tid`	Specifies the task ID of the task whose notepad register will be read. If `tid` equals 0, then the calling task reads its own notepad register.
`regnum`	Specifies the register number. Registers numbered 0 through 7 are for application use, and registers 8 through 15 are reserved for system purposes.
`reg_value`	Points to the variable where `t_getreg()` stores the register's contents.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The kernel does not deny access to those registers reserved for system use. For future compatibility, however, you should not use them.

Multiprocessor Considerations

If the `tid` identifies a global task residing on another processor node, the local kernel will internally make an RSC to that remote node to obtain the register's content for that task.

Callable From

- Task.
- ISR, if the task is local to the node from which the `t_getreg()` call is made.
- KI, if the task is local to the node from which the `t_getreg()` call is made.
- Callout, if the task is local to the node from which the `t_getreg()` call is made.

See Also

`t_setreg`

t_ident

Obtains the task identifier of a named task.

```
#include <psos.h>
unsigned long t_ident(
   char name[4],        /* task name */
   unsigned long node,  /* node number */
   unsigned long *tid   /* task ID */
   )
```


Description

This system call enables the calling task to obtain the task ID of a task it knows only by name. This task ID can then be used in all other operations relating to the task.

Most system calls, except `t_create()` and `t_ident()`, reference a task by its task ID. `t_create()` returns the task ID to a task's creator. For other tasks, one way to obtain the task ID is to use `t_ident()`.

Arguments

`name`	Specifies the user-assigned name of the task.
`node`	In multiprocessor systems, is a search order specifier. See *Multiprocessor Considerations*. In a single node system, this argument must be 0.
`tid`	Points to the variable where `t_ident()` stores the task ID of the named task.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Internally, the pSOS+ kernel treats a task name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the task name as a four-byte character array.
2. The pSOS+ kernel does not check for duplicate task names. If duplicate task names exist, a `t_ident` call can return the `tid` of any task with the duplicate name.
3. If `name` is null (for example, `(char*)0`), then the `tid` of the calling task is returned.

Multiprocessor Considerations

1. `t_ident()` converts a task's name to its `tid` using a search order determined by the `node` input parameter, as described in *pSOSystem System Concepts*. Because tasks created and exported by different nodes may not have unique names, the result of this binding may depend on the order in which the object tables are searched.
2. If the master node's Global Object Table must be searched, then the pSOS+m kernel makes a `t_ident()` RSC to the master node.
3. If the task name is null (i.e., `(char*)0`), then the node argument is ignored, and the `t_ident()` operation returns the `tid` of the calling task on the local node.

Callable From

- Task

See Also

t_create

t_info

Queries a Task Object.

```
#include <psos.h>
unsigned long t_info(
   unsigned long tid,              /* Task ID */
   struct tinfo *buf,              /* Object Information buffer */
   )
```


Description

This system call returns information of a specified task. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

tid	Specifies the ID of the Task object being queried.
buf	Contains the Task information.

struct tinfo has the following members.

```
char           name[4];       /* Task name */
unsigned long  flags;         /* Task attributes */
void           (*iip)();      /* T ask's initial starting addr. */
unsigned long  next;          /* ID of the next waiting task */
unsigned short status;        /* Task status.*/
unsigned char  cpriority;     /* Task's current priority.*/
unsigned char  bpriority;     /* Task's base priority */
unsigned char  ipriority;     /* Task's initial priority */
unsigned char  evwantcond;    /* Task's event wait condition */
unsigned short mode;          /* Task's current mode.*/
unsigned short imode;         /* Task's initial mode */
unsigned short amode;         /* Task's ASR mode.*/
unsigned long  tslice_quantum;/* Per task's time slice in ticks*/
long           tslice_remain; /* Remainder time slice in ticks.*/
unsigned long  wtobid;        /* Object where task is blocked */
unsigned long  evwait;        /* Events - task waiting for.*/
unsigned long  evcaught;      /* Events caught */
unsigned long  evrcvd;        /* Events received */
unsigned long  ss_size;       /* Supervisor stack size */
unsigned long  us_size;       /* User stack size */
unsigned long  *ssp;          /* Current supervisor stack */
                              /* pointer */
unsigned long  *issp;         /* Initial supervisor stack pointer */
```

```
unsigned long  *usp;          /* Current user stack pointer */
unsigned long  *iusp;         /* Initial user stack pointer */
unsigned long  imask;         /* Interrupt priority level */
void           (*asr_addr)(); /* Task's ASR address */
unsigned long  signal;        /* Asynchronous signals pending */
unsigned long  xdate;         /* Timer expiry date */
unsigned long  xtime;         /* Timer expiry time */
unsigned long  xticks;        /* Timer expiry ticks */
unsigned long  nrnunits;      /* No. of region units wanted */
void           **tsdp;        /* Task_specific_Data pointer */
unsigned long  co_toproc;     /* Callouts to process */
unsigned long  co_inprog;     /* Callouts in progress */
unsigned long  evasr_ntfy;    /* ev used for ASR notification */
```

The name of this task is returned in **name**.

The attributes with which this task was created are in **flags**. **iip** specifies the task start address. **next** has the ID of the next waiting task on an object wait queue. When the task is not waiting on any resource, **next** is invalid. If the pSOS operating system is not a 2.5 or a later release, **next** returns 0 when the waiting task is an agent task (in the case of pSOS+m), otherwise **next** represents the remote task ID.

The run time state of the task is present in **status**. Task status could be any of the following.

```
TS_DEBUG       0x8000         /* Blocked by debugger */
TS_SUSP        0x4000         /* Suspended */
TS_TIMING      0x1000         /* Being timed */
TS_PAUSE       0x0800         /* Task paused */
TS_ABSTIME     0x0400         /* Timed for absolute time */
TS_VWAIT       0x0200         /* Waiting for events */
TS_SWAIT       0x0080         /* Waiting for semaphore */
TS_MWAIT       0x0040         /* Waiting for memory */
TS_QWAIT       0x0020         /* Waiting for message */
TS_RWAIT       0x0010         /* Waiting for reply packet */
TS_MUWAIT      0x0008         /* Waiting for mutex */
TS_CVWAIT      0x0004         /* Waiting for condition variable */
TS_CWAIT       0x0001         /* Waiting for component resource */
TS_READY       0x0000         /* Task is in Ready State */
```

A task could be suspended (TS_SUSP bit on), ready to run (TS_READY bit on), blocked by a debugger (TS_DEBUG bit on), or waiting on a resource and/or with a timeout. The task can wait at just one resource (one of the bits, TS_VWAIT, TS_SWAIT, TS_MWAIT, TS_QWAIT, TS_RWAIT, TS_MUWAIT, TS_CVWAIT, TS_CWAIT is on) at a time. If at the same time, it was waiting with a finite timeout, the TS_TIMIMG bit would be on and **xticks** would have the value of the timeout in number of ticks. If the task is waiting for the resource indefinitely, then the TS_TIMIMG bit would be off.

If a task is waiting only with timeouts (not on a resource) then TS_TIMING would be on. Also, if the TS_PAUSE and TS_ABSTIME bits are on, then the task is waiting for the arrival of a future absolute time specified in date, time, ticks, format. These would be stored in **xdate**, **xtime**, and **xticks** respectively. If only TS_TIMING and TS_PAUSE are turned on, then the task is waiting with a timeout value specified in **xticks**. The task will wake up after that number of ticks have elapsed.

cpriority, bpriority, ipriority represent task's current, base and initial priorities respectively. **mode** and **imode** specify task's current and initial modes.

Value in **tslice_remain** is the tasks' remainder in time ticks in case of round robin scheduling. The task's original quantum is given in **tslice_quantum**.

The object id where the task is blocked is given in **wtobid**. This is valid only when the task is blocked on a resource which has an ID in the pSOS object name space. If the task status is either T_MUWAIT (waiting on a mutex) or T_CVWAIT (waiting on a condition variable) and wtobid is -1, then it means the task is waiting on an embedded mutex or embedded condition variable respectively.

The event wait condition is returned in **evwantcond**. This is the value of flags passed to a previous ev_recv() call. The events for which the task may be waiting are given in **evwait**. **evrcvd** specify the events received and pending. These are the events the task is not currently waiting for. **evcaught** specify the actual wanted events that are received. All these are valid only when the task is waiting on an event. 0 is returned in all other cases.

ss_size and **us_size** specify the sizes of supervisory stack and user stack respectively. **ssp** and **issp** specify the task's current and initial supervisory stack addresses. **usp** and **iusp** specify the task's current and initial user stack addresses. A value of 0xDEADDEAD for **iusp** indicates, there is no user stack. when t_info() is called on the running task, **usp** and **ssp** are invalid.

imask specifies task's current interrupt priority mask.

The value in **amode** specifies the ASR mode of the task. **signal** specifies the asynchronous signals currently pending. **asr_addr** is the task's ASR address. Whenever an ASR is posted to this task, it is notified by the sending of an event signal **evasr_ntfy**, if non-zero.

When the task is waiting for memory (i.e. status is set to T_RNWAIT), **nrnunits** specify the memory in region units, the task is requesting for. The size of the region unit can be obtained from calling rn_info() on **wtobid**.

tsdp is the address of the array of the task's task-specific data pointers.

co_toproc and **co_inprog** are the task's callout-related flags, which could be a logical combination of the three callout types, viz., CO_START, CO_RESTART, and CO_DELETE, defined in **psos.h**. If the flag contains a particular callout type, then **co_toproc** denotes that the task has callouts of that type pending invocation, while **co_inprog** denotes that the task has callouts of that type in progress.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. You should set the value of the `SC_PSOS_QUERY` macro to `YES` in the file `<sys_conf.h>` to access the pSOS+ query calls.
2. On certain platforms the task user and supervisor stacks are actually a single stack whose size is the sum of the sizes of the two stacks you specified at the time of task creation. On such platforms the **ss_size** field of the **tinfo** structure would return the combined size of supervisor and user stacks and the us_size field will contain zero.

Multiprocessor Considerations

`t_info()` can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

t_create, *rn_info*, *ev_receive*, *co_register*, *tm_set*

t_mode

Gets or changes the calling task's execution mode.

```
#include <psos.h>
unsigned long t_mode(
   unsigned long mask,       /* attributes to be changed */
   unsigned long new_mode,   /* new attributes */
   unsigned long *old_mode   /* prior mode */
   )
```


Description

This system call enables a task to modify certain execution mode fields. These are preemption on/off, roundrobin on/off, asynchronous signal handling on/off, and interrupt control.

Preemption has precedence over timeslicing. Therefore, if preemption is off, timeslicing does not occur whether or not it is set.

The calling task can be preempted as a result of this call, if its preemptibility is turned from off to on and a higher priority task is ready to run.

To obtain a task's current execution mode without changing it, use a `mask` of 0.

Arguments

`mask`	Specifies all task attributes to be modified.
`new_mode`	Specifies the new task attributes.
`old_mode`	Points to the variable where `t_mode()` stores the old value of the task's mode. If `old_mode` is a NULL address, the task's old mode is not returned.

You create the arguments `mask` and `new_mode` by ORing symbolic constants from the pairs below. These symbolic constants are also defined in `psos.h`.

`T_PREEMPT`	Task is preemptible.
`T_NOPREEMPT`	Task is non-preemptible.
`T_TSLICE`	Task can be time-sliced.
`T_NOTSLICE`	Task cannot be time-sliced.
`T_ASR`	Task's ASR is enabled.
`T_NOASR`	Task's ASR is disabled.

T_ISR	Hardware interrupts are enabled while the task runs.
T_NOISR	Hardware interrupts are disabled while the task runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.
T_LEVELMASK0 through T_LEVELMASK*n*	Certain hardware interrupts are disabled while the task runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.

To create the argument new_mode, you pick symbolic constants from the pairs described above. For instance, to specify that a task have preemption turned off, you place the symbolic constant T_NOPREEMPT in new_mode. To specify that the task have preemption turned off and roundrobin by time-slicing turned on, you place both T_NOPREEMPT and T_TSLICE in new_mode, using the following syntax:

```
T_NOPREEMPT | T_TSLICE
```

The argument mask specifies the bit mask used to permit attribute modifications, and as such, it must contain *both* of the symbolic constants from each pair whose attribute is to be modified. For instance, to enable modification of preemption mode, you place both T_PREEMPT and T_NOPREEMPT in mask. To enable modification of both preemption mode and roundrobin mode, you place both symbolic constants from both pairs in mask, as below:

```
T_PREEMPT | T_NOPREEMPT | T_NOTSLICE | T_TSLICE
```

Target

Interrupt Control

Interrupt control means that while a task is executing, hardware interrupts are disabled. On some processors, you can disable all interrupts at or below a certain interrupt level and enable all interrupts above that level. On other processors you can

simply specify that all interrupts are either enabled or disabled. Details are provided below:

68K CF 960

On 68K, ColdFire, and 960 processors, you can specify the hardware interrupt levels to disable. For instance, on 960 processors, where interrupts have 32 different levels, you could specify that all interrupts at level 15 or below are disabled. Note that interrupt levels have no relation to task priority levels.

Disabling certain interrupt levels is called masking. You set the interrupt mask level by placing the symbolic constant T_LEVELMASK*n* in the mask argument, where *n* is the highest available interrupt level, and placing any symbolic constant between T_LEVELMASK0 and T_LEVELMASK*n* in the new_mode argument. The constant in new_mode indicates the highest interrupt level to mask.

On 68K and ColdFire processors, the highest interrupt level is 7 and the lowest is 0. On 960 processors, the highest interrupt level is 31 and the lowest is 0.

PPC x86 MIPS ARM

On PowerPC, x86, MIPS and ARM processors, you can simply enable or disable all hardware interrupts. To do this, you place both T_ISR and T_NOISR in the mask argument and place either T_ISR or T_NOISR in the new_mode argument.

NOTE: t_mode() stores in old_mode the previous setting of the interrupt control value as stored in the task's TCB (called the true mode), rather than the value in the task's processor status register (called the *transient* mode.) These two modes are normally the same. The one instance when they can be different is if the task changes the interrupt control value without using t_mode().

Processor Mode

`t_mode()` cannot modify the task's processor mode. Most processors only have one processor mode, so this is not relevant. Exceptions are handled as follows:

68K CF ARM PPC

`t_mode()` stores in `old_mode` the previous processor mode (user or supervisor) as stored in the task's TCB (the true mode), rather than the value in the task's processor status register (the transient mode.) These two modes are normally the same, but two exceptions exist:

- A user mode task is in supervisor mode while executing a device driver.
- The task changes to the supervisor mode by executing a trap instruction.

Return Value

This system call always returns 0.

Error Codes

None.

Notes

Multiprocessor Considerations

None. Because `t_mode()` affects only the calling task, its action stays on the local node.

Callable From

- Task

See Also

`t_start`

t_restart

Forces a task to start over regardless of its current state.

```
#include <psos.h>
unsigned long t_restart(
   unsigned long tid,      /* task identifier */
   unsigned long targs[4]  /* startup arguments */
   )
```


Description

This system call forces a task to resume execution at its original start address regardless of its current state or place of execution. If the task was blocked, the pSOS+ kernel forcibly unblocks it. The task's priority and stacks are set to the original values that `t_create()` specified. Its start address and execution mode are reset to the original values established by `t_start()`. Any pending events, signals, or armed timers are cleared. Thereafter, pSOS+ posts any "task restart callouts" that are dispatched in the order they were registered when the task is scheduled to run. All of the "task restart callouts" so dispatched run to completion before the task regains control at its starting address.

The `t_restart()` call accepts a new set of up to four arguments, which, among other things, can be used by the task to distinguish between the initial startup and subsequent restarts.

The calling task does not have to be the creator (or parent) of the task it restarts. However, a task must be restarted from the node on which it was created.

Arguments

`tid`	Specifies the task to restart. When `tid` equals 0, the calling task restarts itself.
`targs`	Specifies up to four long words of input that are passed to the restarted task.

Target

Startup Values

At the start of the task, the CPU registers and the stack are initialized in such a way that if the outermost function of the task exits, the task is deleted; and if that fails,

the task is suspended. Any subsequent attempts to resume the task's execution would result in an illegal address error.

A restarted task can receive up to four long words of input arguments. To facilitate retrieval of these arguments, they are passed to the task as if it is invoked as a high-level language procedure or function. For example, if a C task `nice` has three input arguments, it can be declared as follows:

```
nice (unsigned long a, unsigned long b, unsigned long c);
```

where `targs[0]`is passed to `a`, `targs[1]`to `b`, and `targs[2]` to `c`. In this case, `targs[3]`is irrelevant and does not need the calling task to load it.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Even though `t_restart()` resets the task's stacks, the stacks' contents remain intact. The stack frame and any automatic variables it contains for the task's outermost procedure should remain intact (despite restart) until something is stored into it, or it is initialized.
2. All the mutexes owned by the task are unlocked when the task gets restarted. The calling task may get preempted as a result of `t_restart()` call.

Multiprocessor Considerations

None. A task can be restarted from the local node only.

Callable From

- Task

See Also

`t_create`, *`t_start`*, *`t_delete`*, *`co_register`*

t_resume

Resumes a suspended task.

```
#include <psos.h>
unsigned long t_resume(
   unsigned long tid   /* task identifier */
   )
```


Description

This system call removes the suspension of a task. If the task was suspended while in the ready state, `t_resume()` releases it to be scheduled for execution. If the task was both suspended and blocked (for example, waiting for a message), `t_resume()` removes only the suspension. This leaves the task in the blocked state.

Arguments

`tid` Specifies the task ID of the task.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The calling task can be preempted as a result of this call.

Multiprocessor Considerations

If `tid` identifies a global task residing on another processor node, the local kernel internally makes an RSC to that remote node to resume the task.

Callable From

- Task.
- ISR, if the task is local to the node from which the `t_resume()` call is made.

- KI, if the task is local to the node from which the `t_resume()` call is made.
- Callout, if the task is local to the node from which the `t_resume()` call is made.

See Also

`t_suspend`

t_setpri

Optionally gets and changes a task's priority.

```
#include <psos.h>
unsigned long t_setpri(
   unsigned long tid,       /* task identifier */
   unsigned long newprio,   /* new priority */
   unsigned long *oldprio   /* previous priority */
   )
```


Description

This system call enables the calling task to optionally obtain and modify either its own or another task's base scheduling (software) priority. If `oldprio` is a non-NULL address, the previous base scheduling priority is returned in the variable pointed to by `oldprio`, regardless of the task's current priority. If the calling task issues `t_setpri()` for a target task (self or another) with a valid non-0 value of `newprio`, then the target task (whether it is running or not running) shall resume execution after all other runnable tasks of equal or greater priority have been scheduled to run.

Arguments

`tid`	Specifies the selected task for the priority change. If `tid` equals 0, the calling task is the target.
`newprio`	Specifies the task's new priority. `newprio` must be between 0 and 255 (see Note). If `newprio` is 0, the task's priority is not changed. This allows a read of a task's priority without changing its priority.
`oldprio`	Points to the variable where `t_setpri()` stores the task's previous priority. If `oldprio is a NULL address, the task's original priority is not returned.`

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the calling task uses `t_setpri()` to lower its own priority, it can be preempted by a ready task with higher or equal priority.
2. If the calling task uses `t_setpri()` to raise the priority of another task, it can be preempted if that task is ready and now possesses higher priority.
3. If the calling task uses `t_setpri()` to change its priority to its original one, it can be preempted by another ready task with the same priority.
4. Priority level 0 is reserved for the pSOS+ daemon task IDLE. Priority levels 240 - 255 are reserved for a variety of high priority pSOSystem daemon tasks. While `t_create()` will allow creation of tasks at these priorities, there should never be a need to do so.
5. A task's current priority may be different from its base priority if it owns one or more mutexes. If the task's current priority is less than its new base priority, its current priority is also raised to the new base priority. Otherwise, the task's current priority is not changed. The value returned through `oldprio` is always the base priority of the task regardless of the task's current priority. Hence, the value returned through `oldprio` may not reflect a task's current priority.
6. If the target task owns one or more mutexes, is ready, and its original priority is changed, it shall be scheduled to run before all other ready tasks with equal priority.

Multiprocessor Considerations

If the `tid` identifies a global task residing on another processor node, the local kernel internally makes an RSC to that remote node to change the priority of the task.

Callable From

- Task

See Also

`t_create`, *`mu_lock`*

t_setreg

Sets a task's notepad register.

```
#include <psos.h>
unsigned long t_setreg(
   unsigned long tid,      /* task identifier */
   unsigned long regnum,   /* register number */
   unsigned long reg_value /* register value */
   )
```


Description

This system call enables the caller to modify the contents of a task's notepad register. Each task has 16 such software registers, held in the task's TCB. The purpose of these registers is to furnish every task with a set of named, permanent variables. Eight of these registers are reserved for system use, and eight are free to be used for application-specific purposes.

Arguments

`tid`	Specifies the task ID of the task whose notepad register is set. If `tid` equals 0, the calling task sets its own notepad register.
`regnum`	Specifies the register number. Registers 0 through 7 are for application use, and registers 8 through 15 are reserved for system use.
`reg_value`	Specifies the value at which the register's contents are to be set.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

The kernel does not deny access to the registers that the system reserves. For future compatibility, however, avoid using these reserved registers.

Multiprocessor Considerations

If `tid` identifies a global task residing on another processor node, the local kernel internally makes an RSC to that remote node to set the register for the task.

Callable From

- Task.
- ISR, if the task is local to the node from which the `t_setreg()` call is made.
- KI, if the task is local to the node from which the `t_setreg()` call is made.
- Callout, if the task is local to the node from which the `t_setreg()` call is made.

See Also

`t_getreg`

t_start

Starts a task.

```
#include <psos.h>
unsigned long t_start(
   unsigned long tid,      /* task identifier */
   unsigned long mode,     /* initial task attributes */
   void (*start_addr)(),   /* task address */
   unsigned long targs[4]  /* startup task arguments */
   )
```


Description

This system call places a newly created task into the ready state to await scheduling for execution. Thereafter, pSOS+ posts any "task startup callouts" that are dispatched in the order they were registered when the task is scheduled to run. All of the "task startup callouts" so dispatched run to completion before the task regains control at its starting address. The calling task does not have to be the creator (or parent) of the task to be started. However, a task must be started from the node on which it was created.

Arguments

`tid` Specifies the task to start. `tid` is returned by the `t_create()` and `t_ident()` calls.

`mode` Specifies the initial task attributes. `mode` is formed by ORing the following symbolic constants (one from each pair), which are defined in `<psos.h>`. For instance, to specify that a task should have preemption turned off, you place the symbolic constant `T_NOPREEMPT` in `mode`. To specify that the task should have preemption turned off and roundrobin by time-slicing turned on, you place both `T_NOPREEMPT` and `T_TSLICE` in `mode`, using the following syntax:

```
T_NOPREEMPT | T_TSLICE
```

`T_PREEMPT`	Task is preemptible.
`T_NOPREEMPT`	Task is non-preemptible.
`T_TSLICE`	Task can be time-sliced.
`T_NOTSLICE`	Task cannot be time-sliced.
`T_ASR`	Task's ASR is enabled.
`T_NOASR`	Task's ASR is disabled.

	T_USER	Task runs in user mode.
	T_SUPV	Task runs in supervisor mode. (See *User and Supervisor Modes* under *Target*.)
	T_ISR	Hardware interrupts are enabled while task runs.
	T_NOISR	Hardware interrupts are disabled while task runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.
	T_LEVELMASK0 through T_LEVELMASK*n*	Certain hardware interrupts are disabled while the task runs. These options are available only on certain processors. See *Interrupt Control* under *Target*.
start_addr	Specifies the task's location in memory.	
targs	Specifies four startup values passed to the task (see *Startup Values* under *Target*).	

Target

Startup Values

At the start of the task, the CPU registers and the stack are initialized in such a way that if the outermost function of the task exits, the task is deleted; and if that fails, the task is suspended. Any subsequent attempts to resume the task's execution would result in an illegal address error.

A new task can receive up to four long words of input arguments. To facilitate retrieval of these arguments, they are passed to the task as if it is invoked as a high-level language procedure or function. For example, if a C task `nice` has three input arguments, it can be declared as follows:

```
nice (unsigned long a, unsigned long b, unsigned long c);
```

where `targs[0]` is passed to `a`, `targs[1]` to `b`, and `targs[2]` to `c`. In this case, `targs[3]` is irrelevant and does not need the calling task to load it.

User and Supervisor Modes

You use the symbolic constants T_USER and T_SUPV on each processor as follows:

PPC 68K ARM CF

On PowerPC, 68K, ARM, and ColdFire processors, you must place one of the symbolic constants T_USER or T_SUPV in mode to specify the task's processor mode.

960 MIPS x86

On 960, MIPS and x86 processors, a task can execute only in supervisor mode. In <psos.h>, for these processors, the symbols T_SUPV and T_USER are defined as:

```
#define   T_SUPV   0
#define   T_USER   0
```

for the sole purpose of compatibility with platforms that support both user and supervisor modes.

Interrupt Control

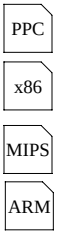
Interrupt control means that while a task is executing, hardware interrupts are disabled. On some processors, you can disable all interrupts at or below a certain interrupt level and enable all interrupts above that level. On other processors you can simply specify that all interrupts are either enabled or disabled. Details are provided below:

68K CF 960

On 68K, ColdFire, and 960 processors, you can specify the hardware interrupt levels to disable. For instance, on 960 processors, where interrupts have 32 different levels, you could specify that all interrupts at level 15 or below are disabled. Note that interrupt levels have no relation to task priority levels.

Disabling certain interrupt levels is called masking. You set a task's interrupt mask level by placing any symbolic constant between T_LEVELMASK0 and T_LEVELMASK*n* in the mode argument, where *n* is the highest available interrupt level.

On 68K and ColdFire processors, the highest interrupt level is 7 and the lowest is 0. On 960 processors, the highest interrupt level is 31 and the lowest is 0.



On PowerPC, x86, ARM and MIPS processors, you can simply enable or disable all hardware interrupts. You do this by placing either `T_ISR` or `T_NOISR` in the `mode` argument.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Once started, the task can compete for the CPU and all other system resources. Thus, it can preempt the calling task if it has a higher priority.
2. `t_start()` calls the optional user-supplied callout procedure whose address is defined by entry `kc_startco` in the pSOS+ Configuration Table.

Multiprocessor Considerations

None. A task can only be started from the local node only.

Callable From

- Task

See Also

`t_create`, *`t_restart`*, *`t_ident`*, *`co_register`*

t_suspend

Suspends a task indefinitely.

```
#include <psos.h>
unsigned long t_suspend(
   unsigned long tid   /* task identifier */
   )
```


Description

This system call suspends execution of a task until a `t_resume()` call is made for the suspended task. The calling task suspends either itself or another task. The `t_suspend()` call prevents the specified task from contending for CPU time but does not directly prevent contention for any other resource. See Note 4.

Arguments

`tid`	Specifies the task to suspend. If `tid` equals zero, the calling task suspends itself.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A task that calls `t_suspend()` on itself always returns 0.
2. A suspended task can be deleted.
3. `t_resume()` is the only call that reverses a suspension.
4. A task can be suspended in addition to being blocked. For example, if a task is waiting for a message at a queue when suspension is ordered, suspension continues after a message has been received. For another example, consider a task P that is blocked while it waits for an event. Another task Q decides that P must not run, and therefore Q suspends P. When P receives the event, it must still

wait for a resumption before it can be ready to run. On the other hand, if Q resumes P while P is still waiting for its event, P continues to wait for the event.

Multiprocessor Considerations

If `tid` identifies a global task residing on another processor node, the local kernel internally makes an RSC to the remote node to suspend the task.

Callable From

- Task

See Also

t_resume

t_tslice

Gets and optionally changes its own or another task's timeslice quantum.

```
#include <psos.h>
unsigned long t_tslice(
   unsigned long tid,          /* task identifier */
   unsigned long new_tslice,   /* new timeslice value*/
   unsigned long *old_tslice   /* old timeslice value */
   )
```


Description

This system call enables the calling task to obtain and optionally modify its own or another task's timeslice quantum. It allows applications to control how much execution time is distributed among tasks with the same priority. It also returns the previous timeslice quantum.

Arguments

`tid`	Specifies the task ID of the selected task for the timeslice quantum change. If tid equals 0, the calling task is setting its own timeslice.
`new_tslice`	Specifies the number of clock ticks to be used for the task's new timeslice value. Its value of must be less than (2^{16}-1). If the value is set to zero, the specified task's timeslice quantum is not changed. This allows the user to read a task's timeslice quantum without changing it.
`old_tslice`	Points to the variable where `t_tslice()` stores the previous timeslice quantum of the specified task. If `old_tslice` is NULL, the previous timeslice quantum will not be returned.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

Multiprocessor Considerations

None. An application may only access the timeslice of tasks on the local node.

Callable From

- Task.

See Also

tm_wkafter, *t_mode*

tm_cancel Cancels an armed timer.

```
#include <psos.h>
unsigned long tm_cancel(
   unsigned long tmid    /* timer identifier */
   )
```


Description

This system call enables a task to cancel a timer armed previously by `tm_evafter()`, `tm_evwhen()`, or `tm_evevery()`. The timer must have been armed by the calling task.

Arguments

`tmid` Specifies the timer to cancel.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

When a task with armed timers is restarted or deleted, its timers are automatically cancelled.

Multiprocessor Considerations

None. This call affects only the calling task.

Callable From

- Task

See Also

tm_evafter, *tm_evevery*, *tm_evwhen*

tm_evafter

Sends events to the calling task after a specified interval.

```
#include <psos.h>
unsigned long tm_evafter(
   unsigned long ticks,   /* delay */
   unsigned long events,  /* event list */
   unsigned long *tmid    /* timer identifier */
   )
```


Description

This system call enables the calling task to arm a timer so that it expires after the specified interval, at which time the pSOS+ kernel internally calls `ev_send()` to send the designated events to this task. Unlike `tm_wkafter()`, `tm_evafter()` does not block the caller. A task can use multiple `tm_evafter()` calls to arm two or more concurrent timers.

The timer interval is specified in system clock ticks. For example, if the system clock frequency is 60 ticks per second and the caller requires a timer to interrupt in 20 seconds, the input specification should be 60x20 (ticks=1200). A timer interval of n ticks causes the events to be sent on the nth next tick. Because `tm_evafter()` can happen anywhere between two ticks, the actual interval is between n-1 and n ticks.

Arguments

`ticks`	Specifies the timer interval.
`events`	Specifies the events to deliver upon expiration of the timer interval. The events are encoded into a long word with bits 31-16 reserved for system use and bits 15 - 0 available for application use.
`tmid`	Points to the variable where `tm_evafter()` stores a timer identifier, which can be used if the armed timer must be cancelled.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The maximum interval is 2^{32}-1 ticks.
2. The timer is counted down by successive `tm_tick()` calls. If no clock or timer is provided, a timer does not expire.
3. A task must call `ev_receive()` explicitly to receive any timer-triggered events (which are like other events in every other way).
4. A task with active timers can be blocked or suspended. In either case, the designated events are sent when the timer expires.
5. When a task with armed timers is restarted or deleted, its timers are automatically cancelled.
6. The number of simultaneously active timers is fixed and defined by the `kc_ntimer` entry in the pSOS+ Configuration Table.

Multiprocessor Considerations

None. This call only affects the calling task.

Callable From

- Task

See Also

`ev_receive`, *`ev_send`*

tm_evevery Sends events to the calling task at periodic intervals.

```
#include <psos.h>
unsigned long tm_evevery(
   unsigned long ticks,   /* delay */
   unsigned long events,  /* event list */
   unsigned long *tmid    /* timer identifier */
   )
```


Description

This system call is similar to `tm_evafter()` except that the armed timer expires periodically instead of once. Events are generated at the specified interval until the timer is cancelled with `tm_cancel()`.

The `tm_evevery()` call provides a drift-free mechanism for performing an operation at periodic intervals. Like `tm_evafter()`, the interval is specified in system clock ticks.

Arguments

`ticks`	Specifies the periodic interval in system clock ticks.
`events`	Specifies the events to deliver upon expiration of the timer interval. The events are encoded into a long word with bits 31-16 reserved for system use and bits 15 - 0 available for application use.
`tmid`	Points to the variable where `tm_evevery()` stores a timer identifier, which can be used if the armed timer must be cancelled.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The maximum interval is 2^{32}-1 ticks.
2. A timer is counted down by successive `tm_tick()` calls. If no clock or timer is provided, a timer does not expire.
3. A task must explicitly call `ev_receive()` to receive timer-triggered events (which are like other events in every other way).
4. A task with active timers can be blocked or suspended. In either case, the designated events are sent when the timer expires.
5. When a task with armed timers is restarted or deleted, its timers are automatically cancelled.
6. The number of simultaneously active timers is fixed and defined by the `kc_ntimer` entry in the pSOS+ Configuration Table.

Multiprocessor Considerations

None. This call affects only the calling task.

Callable From

- Task

See Also

`ev_receive`, *`ev_send`*

tm_evwhen

Sends events to the calling task at a specified time.

```
#include <psos.h>
unsigned long tm_evwhen(
   unsigned long date,   /* date of wakeup */
   unsigned long time,   /* time of wakeup */
   unsigned long ticks,  /* ticks at wakeup */
   unsigned long events, /* event list */
   unsigned long *tmid   /* timer identifier */
   )
```


Description

This system call enables the calling task to arm a timer so that it expires at the appointed date and time, whereupon the pSOS+ kernel internally calls ev_send() to send the designated events to the task. tm_evwhen() does not block the calling task (unlike tm_wkwhen()). A task can use multiple tm_evwhen() calls to arm two or more concurrent timers.

The tm_evwhen() call resembles tm_evafter() except that tm_evwhen() wakes the caller at an appointed time rather than after a specified interval.

Arguments

date — Specifies the clock date for event send. date is encoded as follows:

Field	Bits
Year, A.D.	31-16
Month (1-12)	15-8
Day (1-31)	7-0

time — Specifies the clock time for event send. time is encoded as follows:

Field	Bits
Hour (0-23)	31-16
Minute (0-59)	15-8
Second (0-59)	7-0

`ticks`	An optional count that begins after the last second of time has elapsed. This parameter provides a finer resolution of time, if needed.
`events`	Specifies the events to deliver upon expiration of the timer. The events are encoded into a long word with bits 31-16 reserved for system use and bits 15 - 0 available for application use.
`tmid`	Points to the variable where `tm_evwhen()` stores a timer identifier, which can be used if the armed timer must be cancelled.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A timer is counted down by successive `tm_tick()` calls. If no clock or timer is provided, a timer does not expire.
2. A timer established by `tm_evwhen()` is affected by a `tm_set()` call if that call changes the date and time. If a `tm_set()` advances the time past a scheduled alarm, it triggers the `ev_send()` immediately. To detect this situation, the application can check the time when it awakens and compare it with the expected time.
3. A task must explicitly call `ev_receive()` to receive a timer-triggered event (which is like any event in every other way).
4. A task with active timers can be blocked or suspended. In either case, the designated events are sent when the timer expires.
5. When a task with armed timers is restarted or deleted, its timers are automatically cancelled.
6. The number of simultaneously active timers is fixed and defined by the `kc_ntimer` entry in the pSOS+ Configuration Table.

Multiprocessor Considerations

None. This call affects the calling task only.

Callable From

- Task

See Also

ev_receive, *ev_send*

tm_get

Obtains the system's current version of the date and time.

```
#include <psos.h>
unsigned long tm_get(
   unsigned long *date,   /* year/month/day */
   unsigned long *time,   /* hour:minute:second */
   unsigned long *ticks      /* ticks */
   )
```

Description

This service call returns the system's current version of the date and time-of-day. If the system timer has not been previously set by `tm_set()`, the returned values are relative to zeros for date, time and ticks at the time when pSOS starts.

Arguments

`date` Points to the variable where `tm_get()` stores the date. `date` is encoded as follows:

Field	Bits
Year, A.D.	31-16
Month (1-12)	15-8
Day (1-31)	7-0

`time` Points to the variable where `tm_get()` stores the time. `time` is encoded as follows:

Field	Bits
Hour (0-23)	31-16
Minute (0-59)	15-8
Second (0-59)	7-0

`ticks` Points to the variable where `tm_get()` stores the number of ticks from the last second of the `time` argument.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The accuracy of the returned date and time depends on the precision of `tm_tick()` activity and the moment that the most recent `tm_set()` call occurred.
2. The algorithm for this call accounts for leap years.

Multiprocessor Considerations

None. This call can only be directed at the local processor node.

Callable From

- Task
- ISR
- KI
- Callout

See Also

`tm_set`, *`tm_tick`*

tm_getticks

Returns the elapsed tick count since the initialization of pSOS+.

```
#include <psos.h>
unsigned long tm_getticks(
   unsigned long *tickshi,  /* high order bits of elapsed */
                            /*tick count */
   unsigned long *tickslo   /* low order bits of elapsed tick
                            /* count */
   )
```

Description

This system call returns the number of ticks elapsed since the initialization of pSOS+.

Arguments

`tickshi`	Points to the variable into which pSOS+ stores the high order 32 bits of an unsigned long 64 bit quantity representing the elapsed tick count.
`tickslo`	Points to a variable into which pSOS+ stores the low order 32 bits of an unsigned 64 bit quantity representing the elapsed tick count.

Return Value

This system call always returns 0.

Error Codes

None.

Notes

None.

Multiprocessor Considerations

None. An application may only read the elapsed tick count on the local node.

Callable From

- ISR
- Task
- KI
- Callout

See Also

tm_tick

tm_set

Sets or resets the system's version of the date and time.

```
#include <psos.h>
unsigned long tm_set(
   unsigned long date,   /* year/month/day */
   unsigned long time,   /* hour:minute:second */
   unsigned long ticks   /* clock ticks */
   )
```

Description

This service call enables a task either to set or reset the system's version of the date and time. If a meaningful date and time are required, this call should be made after each system restart or power-on. Thereafter, the system maintains the date and time based on incoming `tm_tick()` calls and the expected arrival frequency defined in the pSOS+ Configuration Table.

Arguments

`date` Specifies the clock date. `date` is encoded as follows:

Field	Bits
Year, A.D.	31-16
Month (1-12)	15-8
Day (1-31)	7-0

`time` Specifies the clock time. `time` is encoded as follows:

Field	Bits
Hour (0-23)	31-16
Minute (0-59)	15-8
Second (0-59)	7-0

`ticks` Specifies the number of ticks from the last second of the `time` argument.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. This implementation accurately reflects leap years and the current century. For example, the value 0088 means 88 A.D., not 1988 A.D.
2. The pSOS+ kernel maintains a flag that indicates if the system time has been initialized since the last system reboot. Startup clears the flag, and `tm_set()` sets the flag.
3. If the input values are accurate when this call is made, the actual synchronization of the system clock depends on such variables as the execution time of `tm_set()` and the moment it arrives between two ticks. The accuracy is within one or two ticks.
4. `tm_set()` has no effect on tasks that are either timing out or waiting after `tm_wkafter()` or `tm_evafter()` calls because these pause intervals are in clock ticks, not clock time.

Multiprocessor Considerations

None. This call can only be directed at the local processor node.

Callable From

- Task
- ISR

See Also

`tm_get`, *`tm_tick`*

tm_tick

Announces a clock tick to the pSOS+ kernel.

```
#include <psos.h>
unsigned long tm_tick()
```

Description

This system call is used to inform the pSOS+ kernel of the arrival of a new clock tick. The pSOS+ time manager uses it to update its time and date calendar, count down tasks that are timing out, and track a running task's time-slice for round-robin scheduling. Normally, the user's real-time clock ISR calls `tm_tick()`.

The frequency of `tm_tick()` calls is fixed and defined in the pSOS+ Configuration Table as `kc_ticks2sec`. Thus, if the value is 100, the pSOS+ time manager interprets 100 `tm_tick()` calls as one real-time second.

Return Value

This call always returns 0.

Error Codes

None.

Notes

1. `tm_tick()` is very fast: it just notifies the system of the arrival of another clock tick. Most other Time Manager actions that can result from this clock tick are postponed until the pSOS+ kernel dispatches and do not run at the clock interrupt level. This improves deterministic system interrupt response.
2. The system accumulates announced ticks when necessary, so no chance exists for an overrun or missed tick. Typically, the accumulation never counts past 1. However, if a system contains one or more lengthy ISRs that respond to high frequency interrupt sources, they can monopolize the CPU enough to prevent the pSOS+ kernel from processing a tick before the next one arrives. In such rare cases, the pSOS+ kernel accumulates the ticks for subsequent accounting.

Multiprocessor Considerations

None. This call can only be directed at the local processor node.

Callable From

- Task
- ISR
- KI
- Callout

See Also

tm_get, *tm_set*, *tm_getticks*

tm_wkafter

Blocks the calling task and wakes it after a specified interval.

```
#include <psos.h>
unsigned long tm_wkafter(
   unsigned long ticks    /* clock ticks */
   )
```

Description

This system call enables the calling task to block unconditionally for a specified interval. This call resembles self-suspension (`t_suspend(0)`), except that `tm_wkafter()` schedules an automatic resumption after the specified time interval has lapsed. The interval is in system clock ticks. For example, if the system clock frequency is 60 ticks per second and the caller requires a pause of 20 seconds, the input specification should be 60x20 (`ticks`=1200).

Arguments

`ticks` Specifies the number of ticks to elapse during the block.

An interval of n ticks awakens the calling task on the nth next tick. Because `tm_wkafter()` can happen anywhere between two ticks, the actual interval is between n-1 and n ticks.

An interval of 0 ticks has a special function: if no ready tasks have the same priority as the calling (or running) task, the calling task continues. On the other hand, if one or more ready tasks with the same priority as the caller exist, the pSOS+ kernel executes a round-robin by placing the caller behind all ready tasks of the same priority and giving the CPU to one of those tasks. This provides a manual round-robin technique to voluntarily give the CPU to another ready task of the same priority.

Return Value

This call always returns 0.

Error Codes

None.

Notes

1. The maximum interval is 2^{32}-1 ticks.
2. Each successive `tm_tick()` call counts down the specified delay interval. If no clock or timer is provided, the delay interval does not expire.
3. A delayed task can additionally be suspended, and countdown continues regardless. If not cancelled, suspension continues after expiration.
4. A paused task can be deleted.
5. `tm_set()` calls do not affect a pause established by `tm_wkafter()` because the pause counter is not changed (even if the date and time are changed).

Multiprocessor Considerations

None. This call only affects the calling task.

Callable From

- Task

See Also

`tm_tick`, *`tm_wkwhen`*

tm_wkwhen

Blocks the calling task and wakes it at a specified time.

```
#include <psos.h>
unsigned long tm_wkwhen(
   unsigned long date,   /* year/month/day */
   unsigned long time,   /* hour:minute:second */
   unsigned long ticks   /* clock ticks */
   )
```

Description

This call enables the calling task to block unconditionally until a specified time.

The `tm_wkwhen()` call resembles a self-suspension (`t_suspend(0)`), but `tm_wkwhen()` schedules the task to resume at a specified time. This call also resembles `tm_wkafter()`, which awakens the calling task after a specified interval.

Arguments

`date` Specifies the clock date. `date` is encoded as follows:

Field	Bits
Year, A.D.	31-16
Month (1-12)	15-8
Day (1-31)	7-0

`time` Specifies the clock time. `time` is encoded as follows:

Field	Bits
Hour (0-23)	31-16
Minute (0-59)	15-8
Second (0-59)	7-0

`ticks` Specifies the number of ticks within the last second of the `time` argument.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A `tm_set()` call (which changes the date and time) directly affects the wakeup established by `tm_wkwhen()`. If `tm_set()` advances the time past a scheduled wakeup, it triggers the wakeup immediately. If necessary, the application can detect this situation by checking the time when the task awakens and comparing that time to the expected time.
2. A task can be suspended while it waits for wakeup. In this case, the wait continues: if not cancelled, suspension continues after wakeup.
3. A task can be deleted while it waits for wakeup.

Multiprocessor Considerations

None. This call affects only the calling task.

Callable From

- Task

See Also

`tm_tick`, *`tm_wkafter`*

tsd_create

Creates a task-specific data (TSD) object.

```
#include <psos.h>
unsigned long tsd_create(
    char name[4],                   /* Unique name of TSD object */
    unsigned long size,             /* Default size related to TSD */
    unsigned long flags,            /* TSD object attributes */
    void ****tsd_anchorp,           /* Ptr to TSD anchor */
    unsigned long *tsdix            /* Ptr to TSD object index */
    )
```

Description

This system call creates a task-specific data (TSD) object by allocating and initializing a TSD Control Block (TSDCB) according to the specifications supplied with the call.

Task-specific data is supported in pSOS+ kernel by providing each task with an array of pointers. Each pointer should ideally point to a TSD area corresponding to a library, device driver, or other code (called a module hereafter). The array is allocated during task create time, and its address is stored in the Task Control Block of the task. For further discussion, this array will be referred to as the TSD array.

Each module that wants task-specific data creates a TSD object. `tsd_create()`will return an unique index, which the module should save for future reference. This index serves as an identifier for further operations on the TSD object, as well as an index into a task's TSD array to access the TSD area corresponding to the module.

The pSOS+ kernel maintains a system-wide anchor (called TSD anchor hereafter) at a fixed location to store a pointer to the running task's TSD array. The kernel ensures that the TSD anchor is pointing to the currently running task's TSD array, by storing the switched-in task's TSD array address in the TSD anchor on every context switch. The `tsd_create()` call returns the TSD anchor.

Like all objects, a TSD object has an user-assigned name and a kernel-assigned TSD object ID. However, for TSD objects the object ID is internal to the kernel; and the user's handle to a TSD object is the index returned with the `tsd_create()` call. Several flag bits specify the characteristics of the TSD object. The flag bits can specify automatic allocation and initialization of memory to tasks during their creation, for the TSD area corresponding to the TSD object.

Arguments

`name` Specifies the user-assigned name of the new TSD object.

`size` Specifies the default size of the TSD area associated with the TSD object. If `TSD_ALLOC` is set (see *flags*, below), then this is the size of memory block to allocate for the TSD area to a task at its create time. If `TSD_NOALLOC` is set, `size` is ignored.

`flags` Specifies the attributes of the TSD object. `flags` is formed by ORing the following symbolic constants (one from each group), which are defined in `<psos.h>`

`TSD_ALLOC`	Allocate memory at task create time, for TSD area associated with this TSD object, for tasks created after this `tsd_create()` call.
`TSD_NOALLOC`	Do not allocate memory at task creation time for TSD area associated with this TSD object, for any tasks.
`TSD_INIT_CLR`	In combination with `TSD_ALLOC`, this would cause zeroing of the TSD area in the memory allocated to a task being created. When ORed with `TSD_NOALLOC`, this option is ignored.
`TSD_INIT_COPY`	In combination with `TSD_ALLOC`, this would cause copying of data from the parent (creating) task's corresponding TSD area, to the memory allocated to the task being created. When ORed with `TSD_NOALLOC`, this constant is ignored.
`TSD_INIT_NONE`	In combination with `TSD_ALLOC`, this would not initialize the memory allocated to a 'task being created. When ORed with `TSD_NOALLOC`, this constant is ignored.

`tsd_anchorp` Specifies address of location where global TSD anchor is returned.

`tsdixp` Specifies the address of the location where the index identifying this TSD object is returned.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Each module that wants task-specific data will create a TSD object, specifying a name, the size of its task-specific data, and allocation/initialization flags. In the following example, the initialization function `module_init` creates the TSD object by issuing the `tsd_create` call. Upon creation, the module will obtain an unique index `tsdix` as an identifier of the TSD object, that the module should save for future reference. The call will also return the address of the TSD anchor in the variable `tsdanchor`, The code within each module should put all its task-specific data in a structure (`module_tsd`). For every task created after module initialization, memory for the TSD area gets allocated at run-time and its address gets assigned to the appropriate TSD array entry. Now, by dereferencing the TSD anchor to obtain the TSD array, and using `tsdix` to directly index into the array, we can obtain the pointer to the running task's TSD area corresponding to this module. This is shown in the function, `module_routine`.

```
typedef struct {
             int          item1;
             int          *item2;
} module_tsd;
#define      DATASIZE     sizeof(module_tsd)

void         ***tsdanchor;
#define      TSDARRAY     (*tsdanchor)
unsigned long     tsdix;

int       module_init(void)
{
          tsd_create('NAME', DATASIZE, (TSD_ALLOC|TSD_INIT_CLR),
                                        &tsdanchor, &tsdix);
          ...Other initialization....
}

int       module_routine(void)
{
          int     i, *j;
          i = ((module_tsd *)  TSDARRAY[tsdix])->item1;
          j = ((module_tsd *)  TSDARRAY[tsdix])->item2;
          ...Other computation ...
}
```

2. Internally, the pSOS+ kernel treats a TSD object name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the TSD object name as a four-byte character array.

3. The maximum number of TSD objects that can simultaneously exist is defined by the `kc_ntsd` entry in the pSOS+ Configuration Table. It also defines the size of the TSD array belonging to a task, since every TSD object has a corresponding unique index into the array.

4. If `size` is 0, no memory will be allocated at task creation for the task's TSD.

5. If the flags argument has `TSD_ALLOC` set, memory for TSD is allocated only for tasks which are created after the completion of this call. Memory is not allocated for any of the tasks already in the system. So, the task which issues the `tsd_create()` call won't have a TSD area associated with the returned index.

6. If a module wants task-specific data for all tasks in the system, it is advisable for the module to create a TSD object (and reserve an unique index) before tasking begins in the system. For this purpose, the pSOS+ kernel provides a system startup callout routine, which gets called before any tasks (except the IDLE task) get created. All the libraries or user created modules may create TSD objects at this point in time with the automatic allocation option.

7. Memory allocated at task create time for TSD areas associated with TSD objects, is freed at task exit time automatically.

Multiprocessor Considerations

None, TSD objects are local resources.

Callable From

- Task

See Also

tsd_ident, *t_create*, *t_delete*

tsd_delete Deletes a task-specific data (TSD) Object

```
#include <psos.h>
unsigned long tsd_delete(
    unsigned long tsdix,            /* TSD object index */
    )
```

Description

This system call deletes the Task-specific data (TSD) object associated with the specified index, and frees the TSDCB.

The calling task does not have to be the creator of the TSD object in order to issue this call. The call does not free memory allocated as TSD areas, to the caller or any other existing tasks, associated with the TSD object being deleted.

Arguments

`tsdix`	Specifies the index of the TSD object to delete.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Once created, a TSD object is generally used as a mechanism to allocate and initialize data areas to tasks in a controlled manner. Each object can be associated with a component, shared library, or another such module; so that the task can easily access its task-specific data for that module through the corresponding TSD index. The only reason for deleting a TSD object would be to allow the reuse of the TSDCB and index.

2. Memory may have been allocated and assigned as TSD areas to the existing tasks during the existence of the TSD object, either via automatic allocation at task create time or a combination of a dynamic memory allocation and `tsd_setval()` call. `tsd_delete()` does not free this memory; and tasks can

continue accessing their own TSD area via the TSD anchor. However, the user may need to keep track of the inter-relation between every existing task and all the TSD objects that exist or existed during the life-time of the task.

Multiprocessor Considerations

None, `tsd_delete()` can be called on TSD objects created on the local node. All operations on TSD objects are strictly local.

Callable From

- Task

See Also

tsd_create, *tsd_setval*

tsd_getval

Obtains a task's task-specific data (TSD) array entry value associated with the given TSD object.

```
#include <psos.h>
unsigned long tsd_getval(
    unsigned long tsdix,            /* TSD object index */
    unsigned long tid,              /* Task ID */
    void **valp                     /* Address to get value in */
    )
```

Description

This system call enables the calling task to read either its own or another task's TSD area address associated with a particular TSD object. The task's TSD array entry, indexed by the input TSD index, is returned into the memory location provided.

Arguments

`tsdix`	Specifies the index corresponding to the TSD object. It also indexes into the TSD array to determine the entry, whose value is to be set.
`tid`	Specifies the selected task whose TSD array is to be read.
`valp`	Specifies the memory location in which to return the old value of the selected task's TSD array entry.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Multiprocessor Considerations

`tsd_getval()` can be called only on objects created on the local node. It does not initiate RSCs on remote nodes.

Callable From

- Task

See Also

tsd_create, *tsd_setval*

tsd_ident

Obtains the index associated with a named task-specific data (TSD) object.

```
#include <psos.h>
unsigned long tsd_ident(
    char name[4],                   /* TSD object name */
    unsigned long *tsdix,           /* TSD object index */
    )
```

Description

This system call enables the calling task to obtain the index associated with a TSD object it only knows by name. The index can be used to all other operations relating to the TSD object. It can be directly used to index into the running task's TSD array (through the anchor), to obtain the address of the TSD area corresponding to this TSD object.

Arguments

`name`	Specifies the name of the TSD object.
`tsdix`	Specifies the address location to return the index associated with the TSD object in.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Internally, the pSOS+ kernel treats a TSD object name as a 32-bit integer. However, when the application calls the kernel through the pSOS+ C language API, it passes the TSD object name as a four-byte character array.

Multiprocessor Considerations

None, `tsd_ident()` can be called on TSD objects created on the local node. TSD objects can only be local objects.

Callable From

- Task

See Also

`tsd_create`

tsd_info Queries a Task-Specific Data Entry Object.

```
#include <psos.h>
unsigned long tsd_info(
   unsigned long tsdix,          /* Task-specific data object
                                 /*index */
   struct tsdinfo *buf,          /* Object Information buffer */
   )
```

Description

This system call returns information of a specified task-specific data entry object. The information includes the static information that was specified at object creation time and certain internal state information which is not static.

Arguments

tsdix	Specifies the index of the Task-specific data entry (TSD) object being queried.
buf	Contains the TSD information.

struct tsdinfo has the following members.

```
char                  name[4];     /* TSD name */
unsigned long         flags;       /* TSD attributes */
unsigned long         size;        /* Default size of data area */
unsigned long         obid;        /* pSOS-assigned object ID */
```

The name of this TSD is returned in **name**.

The attributes with which this TSD was created are in **flags**. When the TSD is created with the option to automatically allocate data at task-create time, **size** denotes the size of the data area which will be allocated.

A TSD object, like other pSOS+ objects, has a pSOS assigned object ID. However, it also has an additional identifier, which is the TSD index returned with the tsd_create() call. This index serves as an identifier for further operations on the TSD object, as well as an index into any task's TSD array for accessing its corresponding TSD area address. However, a user may want to know the object ID; so it is returned in **obid.**

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

You should set the value of the system configuration parameter, SC_PSOS_QUERY, to YES in the file <sys_conf.h> to access the pSOS+ tsd_info() service calls.

Multiprocessor Considerations

tsd_info() can be called only on objects created on the local node.

Callable From

- Task
- Callout

See Also

tsd_create

tsd_setval

Sets, and optionally gets, a task's data value associated with the given task-specific data (TSD) object.

```
#include <psos.h>
unsigned long tsd_setval(
    unsigned long tsdix,            /* TSD object index */
    unsigned long tid,              /* Task ID */
    void *newval                    /* New value to set */
    )
```

Description

This system call enables the calling task to modify either its own or another task's TSD area address associated with a particular TSD object. The task's TSD array entry, indexed by the input TSD index, is set to the new value. The value of the array entry is returned if the pointer to the old data value is non-zero.

This call is useful in the case when the TSD object has been created without the option of automatic memory allocation during task creation. We can dynamically allocate memory for a task's TSD area, and then use `tsd_setval()` to store the address of the allocated memory into the correct TSD array entry.

Arguments

`tsdix`	Specifies the index corresponding to the TSD object. It also indexes into the TSD array to determine the entry, whose value is to be set.
`tid`	Specifies the selected task whose TSD array is to be changed.
`newval`	Specifies the new value to assign to the TSD array entry of the selected task.

Return Value

This call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A module like shared library may not want to create a TSD object with the automatic allocation option. Such a module will create the TSD object at run-time as in the initialization function, `module_init`, in the example below. Upon creation, the module will obtain an unique index `tsdix` as an identifier of the TSD object, that the module should save for future reference, The library manager can initialize task-specific data for any task it wants by invoking a function like `module_start`, where it dynamically allocates a memory block, and assigns it to the task's TSD array entry using `tsd_setval()`.


```
typedef struct {
                  int           item1;
                  int           *item2;
} module_tsd;
#define           DATASIZE      sizeof(module_tsd)

void              ***tsdanchor, *tsdaddr;
#define           TSDARRAY      (*tsdanchor)
unsigned long           tsdix;

int               module_init(void)
{
                  tsd_create('NAME', 0, TSD_NOALLOC,
                                                &tsdanchor, &tsdix);
                  ...Other initialization....
}

int               module_start(unsigned long tid)
{
                  tsdaddr = malloc (DATASIZE);
                  tsd_setval(tsdix, tid, tsdaddr);
                  ...Other startup ...
}
```

Multiprocessor Considerations

`tsd_setval()` can be called only on objects created on the local node. `tsd_setval()` does not initiate RSCs on remote nodes.

Callable From

- Task

See Also

tsd_create, *tsd_getval*

3 pHILE+ System Calls

This chapter provides detailed information on each system call in the pHILE+ component of pSOSystem. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, the volume types it applies to, a detailed description, its arguments, its return value, and any error codes that it can return. Where applicable, the section also includes the headings "Notes", "Usage", and "See Also". "Notes" contains any important information not specifically related to the call description, "Usage" provides detailed usage information, and "See Also" indicates other calls that have related information.

Structures described in this chapter are also defined in the file `<phile.h>`. Structures must be word-aligned and must not be packed.

If you need to look up a system call by its functionality, refer to Table 3-1 on page 3-2, which lists the calls alphabetically by component and provides a brief description of each call.

Table 3-2 on page 3-4 shows the file systems that each pHILE+ call supports. If a call supports a particular file system, the table entry is "yes." Otherwise, the table entry is the error message produced when a call is either incorrectly used on a file system or attempted on an unsupported file system. Error codes are described in the call descriptions within this chapter, and also in Appendix A, *Error Codes*.

TABLE 3-1 pHILE+ System Calls

Name	Description	Page
access_f	Determines the accessibility of a file.	3-7
annex_f	Allocates contiguous blocks to a file.	3-8
cdmount_vol	Mounts a CD-ROM volume.	3-10
change_dir	Changes the current directory.	3-12
chmod_f	Changes the mode of a named file.	3-14
chown_f	Changes the owner or group of a named file.	3-16
close_dir	Closes an open directory file.	3-17
close_f	Closes an open file connection.	3-17
control_vol	Performs control operations on a volume.	3-20
create_f	Creates a data file.	3-23
fchmod_f	Changes the mode of a file specified by its file identifier.	3-25
fchown_f	Changes the owner or group of a file specified by its file identifier.	3-27
fstat_f	Obtains the status of a file specified by its file identifier.	3-28
fstat_vfs	Obtains statistics about a mounted volume specified by a file identifier.	3-31
ftruncate_f	Changes the size of a file specified by its file identifier.	3-33
get_fn	Obtains the file number of a file.	3-35
init_vol	Initializes a pHILE+ formatted volume.	3-37
link_f	Creates a hard link between two files on the same volume.	3-40
lock_f	Locks or unlocks part or all of an open file.	3-41
lseek_f	Repositions for read or write within an open file.	3-43
lstat_f	Gets the status of a symbolically linked file.	3-45
make_dir	Creates a directory file.	3-47
mount_vol	Mounts a pHILE+ formatted volume.	3-49

TABLE 3-1 pHILE+ System Calls (Continued)

Name	Description	Page
move_f	Moves (renames) a file.	3-51
nfsmount_vol	Mounts a remote file system.	3-53
open_dir	Opens a directory file.	3-55
open_f	Opens a file.	3-56
open_fn	Opens a file by its file identifier.	3-59
pcinit_vol	Initializes an MS-DOS volume.	3-61
pcmount_vol	Mounts an MS-DOS volume.	3-65
read_dir	Reads directory entries in a file system independent format.	3-67
read_f	Reads from a file.	3-69
read_link	Reads the value of a symbolic link.	3-71
read_vol	Reads directly from a pHILE+ formatted volume.	3-73
remove_f	Deletes a file.	3-75
stat_f	Gets the status of a named file.	3-76
stat_vfs	Gets statistics for a named volume.	3-80
symlink_f	Creates a symbolic link to a file.	3-83
sync_vol	Synchronizes a volume.	3-84
truncate_f	Changes the size of a named file.	3-86
unmount_vol	Unmounts a volume.	3-88
utime_f	Sets the access and modification times of a file.	3-90
verify_vol	Verifies a volume's control structures.	3-91
write_f	Writes to an open file.	3-107
write_vol	Writes data directly to a pHILE+ formatted volume.	3-109

TABLE 3-2 File Systems Supported by pHILE+ Calls

Syscall/ Filesystem	pHILE+	MS-DOS	NFS	CD-ROM
access_f	E_FUNC	E_BADMS	yes	E_BADCD
annex_f	yes	E_BADMS	E_BADNFS	E_BADCD
cdmount_vol	E_VALIEN	E_VALIEN	E_MNTED	yes
change_dir	yes	yes	yes	yes
chmod_f	E_FUNC	E_BADMS	yes	E_RO
chown_f	E_FUNC	E_BADMS	yes	E_RO
close_dir	yes	yes	yes	yes
close_f	yes	yes	yes	yes
control_vol	yes	yes	yes	yes
create_f	yes	yes	yes	E_RO
fchmod_f	E_FUNC	E_BADMS	yes	E_RO
fchown_f	E_FUNC	E_BADMS	yes	E_RO
fstat_f	yes	yes	yes	yes
fstat_vfs	yes	yes	yes	yes
ftruncate_f	yes	yes	yes	E_RO
get_fn	yes	yes	E_BADNFS	yes
init_vol	yes	yes	E_MNTED	E_RO
link_f	E_FUNC	E_BADMS	yes	E_BADCD
lock_f	yes	E_BADMS	E_BADNFS	E_BADCD
lseek_f	yes	yes	yes	yes
lstat_f	E_FUNC	E_BADMS	yes	E_BADCD
make_dir	yes	yes	yes	E_RO
mount_vol	yes	E_VALIEN	E_MNTED	E_VALIEN

TABLE 3-2 File Systems Supported by pHILE+ Calls (Continued)

Syscall/ Filesystem	pHILE+	MS-DOS	NFS	CD-ROM
move_f	yes	yes	yes	E_RO
nfsmount_vol	E_MNTED	E_MNTED	yes	E_MNTED
open_dir	yes	yes	yes	yes
open_f	yes	yes	yes	yes
open_fn	yes	yes	E_BADNFS	yes
pcinit_vol	yes	yes	E_MNTED	E_RO
pcmount_vol	E_VALIEN	yes	E_MNTED	E_VALIEN
read_dir	yes	yes	yes	yes
read_f	yes	yes	yes	yes
read_link	E_FUNC	E_BADMS	yes	E_BADCD
read_vol	yes	yes	E_BADNFS	yes
remove_f	yes	yes	yes	E_RO
stat_f	yes	yes	yes	yes
stat_vfs	yes	yes	yes	yes
symlink_f	E_FUNC	E_BADMS	yes	E_BADCD
sync_vol	yes	yes	E_BADNFS	E_RO
truncate_f	yes	yes	yes	E_RO
unmount_vol	yes	yes	yes	yes
utime_f	E_FUNC	E_BADMS	yes	E_RO
verify_vol	yes	E_VALIEN	E_VALIEN	E_VALIEN
write_f	yes	yes	yes	E_RO
write_vol	yes	yes	E_BADNFS	E_RO

access_f

Determines the accessibility of a file.

```
#include <phile.h>
unsigned long access_f(
   char *name,/* file pathname */
   int mode   /* file mode to check */
   )
```


Volume Types

NFS formatted volumes.

Description

`access_f()` checks the named file for accessibility according to `mode`.

Arguments

`name` Points to a null-terminated pathname of a file to be checked.

`mode` Specifies the file mode to check. `mode` is the result of an OR operation performed on the following constants (defined in `<phile.h>`.)

Hex	Mnemonic	Description
4	R_OK	Test for read permission.
2	W_OK	Test for write permission.
1	X_OK	Test for execute/search permission.
0	F_OK	Test for presence of file.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

chmod_f, *stat_f*, *fstat_f*

annex_f

Allocates contiguous blocks to a file.

```
#include <phile.h>
unsigned long annex_f(
   unsigned long fid,         /* file identifier */
   unsigned long alloc_size, /* number of blocks to add */
   unsigned long *blkcount   /* number of blocks added */
   )
```

Volume Types

pHILE+ formatted volumes.

Description

`annex_f()` extends the physical size of a file on a pHILE+ formatted volume by adding a group of contiguous blocks.

Arguments

`fid`	Identifies the file.
`alloc_size`	Specifies the desired number of blocks to add to the file.
`blkcount`	Points to the variable where `annex_f()` stores the number of blocks actually allocated. This number can be less than `alloc_size`, in which case `blkcount` represents the largest group of contiguous blocks available on the volume.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `annex_f()` expands the physical size of a file but does not change its logical size or the end-of-file position.
2. `read_f()` and `lseek_f()` calls into annexed blocks are not allowed until the logical length of the file is extended by writing data into the annexed blocks.
3. A `volume full` error occurs if no blocks can be added to the file.
4. Unless the blocks are merged into the file's last extent, a new extent descriptor is added to the file as a result of an `annex_f()` call.
5. On volumes with separate control and data regions, the pHILE+ file system manager automatically determines the type of block to be annexed (based on the file type.) Directory files receive control blocks, and ordinary files receive data blocks.
6. Annexes to `BITMAP.SYS` and `FLIST.SYS` are not allowed.

See Also

write_f, *open_f*, *read_f*, *lseek_f*

cdmount_vol Mounts a CD-ROM volume.

```
#include <phile.h>
unsigned long cdmount_vol(
   char *device,              /* volume name */
   unsigned long sync_mode    /* synchronization mode */
   )
```

Volume Types

CD-ROM formatted volumes that conform to ISO-9660 specification. Multi-volume sets and interleaved files are not supported.

Description

`cdmount_vol()` mounts a CD-ROM volume. A volume must be mounted before file operations can be applied to it.

Removable volumes can be mounted and unmounted as required. CD-ROM volumes are read-only.

Arguments

`device`	Points to the null-terminated name of the volume to be mounted.
`sync_mode`	Specifies the volume's write synchronization attribute. This attribute is defined in `<phile.h>` and must be set to the value shown below.
	`SM_READ_ONLY` Read-only synchronization mode.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. A CD-ROM volume does not need volume initialization.
2. The number of volumes that can be mounted simultaneously in the system cannot exceed the pHILE+ Configuration Table parameter `fc_nmount`.
3. The pHILE+ file system manager does not attempt verification or any other way of determining volume ownership. Any task can perform a `cdmount_vol()`. A mounted device does not retain a record of the task that mounted it. Therefore, a volume is not automatically unmounted when the task that mounted it is deleted. This and any other security measures, if desired, should be supported by the user's own layer of software.

See Also

mount_vol, *nfsmount_vol*, *pcmount_vol*, *unmount_vol*

change_dir Changes the current directory.

```
#include <phile.h>
unsigned long change_dir(
   char *name   /* directory path */
   )
```

Volume Types

All volume types.

Description

`change_dir()` changes the current directory of the calling task. After `change_dir()` executes, all relative pathnames used by the calling task are relative to the new current directory.

Arguments

`name`	Points to the null-terminated pathname of the new current directory.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The pHILE+ file system manager does not assume a default current directory. Therefore, if a task uses relative pathnames, it must specify the current directory at least once.
2. The input pathname for the new current directory can be a relative pathname. In this case, it is relative to the current directory before the current directory is changed.
3. The pHILE+ file system manager makes no attempt to verify that the current directory corresponds to the intended entities. For example, if the current direc-

tory is deleted or the volume containing the current directory is unmounted, the results of operations by tasks using pathnames relative to the invalid current directory are unpredictable.

See Also

get_fn, *stat_f*

chmod_f

Changes the mode of a named file.

```
#include <phile.h>
unsigned long chmod_f(
   char *name,   /* file pathname */
   int mode      /* new file mode */
   )
```

Volume Types

NFS formatted volumes.

Description

`chmod_f()` changes mode of the named ordinary or directory file.

Arguments

`name` Points to a null-terminated pathname of a file.

`mode` Specifies the new file mode. `mode` is the result of an OR operation performed on the following constants (defined in `<phile.h>`).

Mnemonic	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_ISVTX	Save text image after execution (sticky bit.)
S_IREAD	Read permission, owner.
S_IWRITE	Write permission, owner.
S_IEXEC	Execute/search permission, owner.
S_IRGRP	Read permission, group.
S_IWGRP	Write permission, group.
S_IXGRP	Execute/search permission, group.
S_IROTH	Read permission, other.
S_IWOTH	Write permission, other.
S_IXOTH	Execute/search permission, other.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

fchmod_f, *stat_f*, *fstat_f*, *open_f*, *chown_f*, *fchown_f*

chown_f

Changes the owner or group of a named file.

```
#include <phile.h>
unsigned long chown_f(
   char *name,    /* file pathname */
   int owner,     /* new user ID */
   int group      /* new group ID */
   )
```

Volume Types

NFS formatted volumes.

Description

`chown_f()` changes the owner and group of a file specified by name.

Arguments

`name`	Points to a null-terminated pathname of either an ordinary file or a directory file.
`owner`	Specifies the user ID of the new owner.
`group`	Specifies the group ID of the new group.

User ID and group ID are UNIX terms used to identify a user and a file access group on a UNIX system.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

fchmod_f, *stat_f*, *fstat_f*, *chmod_f*, *fchmod_f*, *open_f*

close_dir

Closes an open directory file.

```
#include <phile.h>
unsigned long close_dir(
   XDIR *dir   /* NFS directory handle */
   )
```


Volume Types

All volume types.

Description

`close_dir()` closes the connection to a directory specified by the directory handle `dir`.

Arguments

`dir`	Points to an `XDIR` structure defined in `<phile.h>`.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

`open_dir`

close_f

Closes an open file connection.

```
#include <phile.h>
unsigned long close_f(
   unsigned long fid   /* file identifier */
   )
```

Volume Types

All volume types.

Description

`close_f()` closes the connection designated by the file identifier `fid`. If `fid` is 0, `close_f()` closes all of the files opened by the calling task. If `close_f()` terminates the last connection to a file, the file's FCB is deallocated.

Arguments

`fid` Specifies the file ID of the file connection to be closed.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Because the total number of open files is limited for both tasks and the system as a whole, `close_f()` should be used whenever a file connection is no longer needed.
2. If the pREPC+ library is in use, `close_f(0)` should be called only after the pREPC+ call `fclose(0)`. Otherwise, files that the pREPC+ library is using can be unexpectedly closed.
3. If the task has opened one or more NFS files, `close_f(0)` must precede any `close(0)` call to the pNA+ network manager by the same task.

See Also

open_f, *open_fn*

control_vol

Performs control operations on a volume.

```
#include <phile.h>
unsigned long control_vol(
char *device,                /* volume name */
unsigned long cmd,           /* command */
void *arg                    /* command argument */
)
```

Volume Types

All volume types. However, not all commands are supported on all volume types as shown in the following table:

Command	Volume type
CONTROL_VOL_CHANGED_VOL	All volume types.
CONTROL_VOL_PCINIT_VOL	MS-DOS formatted volumes.

Description

The `control_vol()` system call is used to perform control operations on the specified volume.

Arguments

`device` Specifies the volume. Some operations require the volume to be mounted, and others not to be mounted.

`cmd` Specifies the operation and is a symbolic constant. All permissible values are defined in <phile.h>.

`arg` Points to a data structure that is dependent on the value of cmd and contains additional information needed by control_vol() to perform the operation.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Operations upon mounted volumes:

   CONTROL_VOL_CHANGED_VOL—Unmount a volume that was changed.

   Optionally it checks whether the volume was changed and unmounts it only if it actually changed. This is done by reading the root block for pHILE+ format, boot record for MS-DOS FAT format, or primary volume descriptor for CD-ROM ISO 9660 format and comparing fields that identify the volume with values remembered when the volume was mounted. If they match the volume was not changed, and is not unmounted.

   For CONTROL_VOL_CHANGED_VOL arg is not a pointer. It is one of two symbolic constants: CONTROL_VOL_ALWAYS_CHANGED, and CONTROL_VOL_IF_CHANGED. The first always unmounts the volume. The second checks whether the volume was changed, and unmounts the volume only if the volume was changed.

   If unmounted, the volume is unmounted slightly differently than unmount_vol(). Cached blocks for the volume are discarded. unmount_vol() writes them to the disk. Open file descriptors on the volume are marked invalid. All operations upon an invalid file descriptor except close_f() or close_dir(), for a file or a directory, respectively, return error E_FIDOFF. After changing a volume the application should close all invalid file descriptors so they can be reused. In contrast, unmount_vol() if there are any open file descriptors on the volume does not unmount the volume but instead returns error E_MNTOPEN.

   control_vol() cmd CONTROL_VOL_CHANGED_VOL is not the preferred way to unmount or change a volume. Unless the volume is mounted SM_IMMED_WRITE any changed blocks cached in memory are lost when control_vol() cmd CONTROL_VOL_CHANGED_VOL is called. If SM_IMMED_WRITE, is used for pHILE+ format, even though all blocks are already written to disk, if the volume has been changed the volume changed bit is not cleared. The preferred way to unmount or change a disk is to first call unmount_vol() for all volumes mounted on the disk, for example, the unpartitioned disk or all mounted partitions, and then remove the disk. If the disk is being changed then insert a new disk, and mount all desired volumes on the disk. This does write all changed cached blocks to disk, and if pHILE+ format, clears the volume changed bit if the volume was changed.

control_vol() cmd CONTROL_VOL_CHANGED_VOL is used to recover from circumstances that are better avoided. It is used with arg CONTROL_VOL_ALWAYS_CHANGED to unmount a disk that cannot be unmounted with unmount_vol() due to unretried I/O errors while writing to the disk the cached changed blocks or volume changed bit. It is used with arg CONTROL_VOL_ALWAYS_CHANGED if a disk is removed without first being unmounted. It is used with arg CONTROL_VOL_IF_CHANGED if a disk is removed and reinserted without first being unmounted.

2. Operations upon not mounted volumes

CONTROL_VOL_PCINIT_VOL—Initialize a MS-DOS FAT format volume.

This is an alternate interface to pcinit_vol(). However, unlike pcinit_vol(), the format parameters are passed as a parameter. This needs to be used to initialize, not reinitialize, a volume in only the two situations below. Otherwise pcinit_vol() can be used.

- If the disk driver does not support DK_DRIVER, to initialize a volume that is a hard disk partition or an unpartitioned disk that is not one of the seven built-in media types.
- To initialize any volume with nonstandard format parameters. One example is initializing a 1.44M 3 1/2" floppy disk in 720K format instead of the standard 1.44M format.

For CONTROL_VOL_PCINIT_VOL arg points to the following structure:

```
#include <phile.h>
typedef struct control_vol_pcinit_arg {
void * scratch_buf;               /* scratch buffer */
DISK_VOLUME_GEOMETRY * geometry;  /* volume geometry */
} CONTROL_VOL_PCINIT_ARG;
```

scratch_buf points to the scratch buffer needed when calling pcinit_vol(). geometry is the volume and file system format parameters. It is of type DISK_VOLUME_GEOMETRY which is in <phile.h>. The fields of DISK_VOLUME_GEOMETRY are explained in Chapter 7 of the *pSOSystem System Concepts* manual.

See Also

pcinit_vol, *unmount_vol*

create_f

Creates a data file.

```
#include <phile.h>
unsigned long create_f(
   char *name,                  /* pathname */
   unsigned long expand_unit,   /* expansion factor */
   unsigned long mode           /* access mode */
   )
```


Volume Types

All volume types (except CD-ROM); however, `expand_unit` is meaningful only on pHILE+ formatted volumes, and `mode` is meaningful only on NFS volumes.

Description

`create_f()` creates a new ordinary file.

Arguments

`name` Points to the null-terminated pathname of the file to create.

`expand_unit` For pHILE+ formatted volumes only, specifies the number of contiguous blocks to add whenever the file is expanded during a `write_f()` system call.

`mode` For NFS volumes only, specifies the access modes associated with the file, and is the result of an OR operation performed on the following constants (defined in <phile.h>).

Mnemonic	Description
S_ISUID	Set user ID on execution
S_ISGID	Set group ID on execution
S_IRUSR	Read permission, owner
S_IWUSR	Write permission, owner
S_IXUSR	Execute/search permission, owner
S_IRGRP	Read permission, group
S_IWGRP	Write permission, group

S_IXGRP	Execute/search permission, group
S_IROTH	Read permission, other
S_IWOTH	Write permission, other
S_IXOTH	Execute/search permission, other

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If a file by the same name already exists, `create_f()` fails. An existing file must first be explicitly deleted using `remove_f()` before the same name can be used for a new file.
2. After a `create_f()` call, the new file is empty. Blocks are allocated as data is written.
3. `create_f()` creates only data files. Use `make_dir()` to create a directory file.
4. `create_f()` does not open a file: use an explicit `open_f()`.
5. For pHILE+ formatted volumes, the input parameter `expand_unit` should be considered carefully because it can affect the data access efficiency of the file.

See Also

`remove_f`, *`make_dir`*, *`open_f`*

fchmod_f

Changes the mode of a file specified by its file identifier.

```
#include <phile.h>
unsigned long fchmod_f(
   unsigned long fid,   /* file identifier */
   int mode             /* new file mode */
   )
```


Volume Types

NFS formatted volumes.

Description

`fchmod_f()` functions the same as `chmod_f()` except that `fchmod_f()` changes the mode of a file by its file identifier instead of its pathname. The file identifier is first obtained with `open_f()`.

Arguments

`fid` Specifies the file identifier associated with the file.

`mode` Specifies the new file mode. `mode` is the result of an OR operation performed on the following constants (defined in `<phile.h>`).

Mnemonic	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_ISVTX	Save text image after execution (sticky bit).
S_IREAD	Read permission, owner.
S_IWRITE	Write permission, owner.
S_IEXEC	Execute/search permission, owner.
S_IRGRP	Read permission, group.
S_IWGRP	Write permission, group.
S_IXGRP	Execute/search permission, group.
S_IROTH	Read permission, other.

S_IWOTH	Write permission, other.
S_IXOTH	Execute/search permission, other.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

chmod_f, *stat_f*, *fstat_f*, *open_f*, *chown_f*, *fchown_f*

fchown_f

Changes the owner or group of a file specified by its file identifier.

```
unsigned long fchown_f(
   unsigned long fid,    /* file identifier */
   int owner,            /* new user ID */
   int group             /* new group ID */
   )
```


Volume Types

NFS formatted volumes.

Description

`fchown_f()` functions the same as `chown_f()` except it changes the `owner` or `group` of a file by its file identifier instead of its pathname. The file identifier is first obtained with `open_f()`.

Arguments

`fid`	Specifies the file identifier associated with the file.
`owner`	Specifies the user ID of the new owner.
`group`	Specifies the group ID of the new group.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

`chmod_f`, *`stat_f`*, *`fstat_f`*, *`chmod_f`*, *`fchmod_f`*, *`open_f`*

fstat_f

Obtains the status of a file specified by its file identifier.

```
#include <phile.h>
unsigned long fstat_f(
   unsigned long fid,   /* file identifier */
   struct stat *buf     /* file status */
   )
```

Volume Types

All volume types.

Description

`fstat_f()` functions the same as `stat_f()` except that `fstat_f()` obtains information about a file by using the file identifier instead of the name. The file identifier is first obtained with either `open_f()` or `open_fn()`.

Arguments

`fid` Specifies the file identifier associated with the file.

`buf` Points to a `stat` structure defined in `<phile.h>`, as follows:

```
struct stat {
   mode_t st_mode;   /* ownership/protection */
   ino_t st_ino;     /* file ID */
   dev_t st_dev;     /* device ID where the volume resides */
   dev_t st_rdev;    /* device ID, for character or
                      * block special files only */
   nlink_t st_nlink; /* number of hard links to the file */
   uid_t st_uid;     /* user ID */
   gid_t st_gid;     /* group ID */
   off_t st_size;    /* total size of file, in bytes */
   time_t st_atime;  /* file last access time */
   time_t st_mtime;  /* file last modify time */
   time_t st_ctime;  /* file last status change time */
   long st_blksize;  /* optimal block size for I/O ops */
   long st_blocks;   /* file size in blocks */
   };
```

`mode_t`, `ino_t`, `dev_t`, `nlink_t`, `uid_t`, `gid_t`, `off_t`, and `time_t` are defined as unsigned long in `<phile.h>`.

The following differences exist for local file systems (pHILE+, MS-DOS, and CD-ROM):

`rdev = dev, nlink = 1, uid = 0, gid = 0, atime = ctime = mtime`

The status information word `st_mode` consists of the following bits:

```
S_IFMT      0170000     /* type of file */
S_IFIFO     0010000     /* fifo special */
S_IFCHR     0020000     /* character special */
S_IFDIR     0040000     /* directory */
S_IFBLK     0060000     /* block special */
S_IFREG     0100000     /* regular file */
S_IFLNK     0120000     /* symbolic link */
S_IFSOCK    0140000     /* socket */
S_ISUID     0004000     /* set user ID on execution */
S_ISGID     0002000     /* set group ID on execution */
S_ISVTX     0001000     /* save swapped text even after use */
S_IRUSR     0000400     /* read permission, owner */
S_IWUSR     0000200     /* write permission, owner */
S_IXUSR     0000100     /* execute/search permission, owner */
S_IRGRP     0000040     /* read permission, group */
S_IWGRP     0000020     /* write permission, group */
S_IXGRP     0000010     /* execute/search permission, group */
S_IROTH     0000004     /* read permission, other */
S_IWOTH     0000002     /* write permission, other */
S_IXOTH     0000001     /* execute/search permission, other */
```


Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

stat_f, *chmod_f*, *fchmod_f*, *chown_f*, *fchown_f*, *link_f*, *read_f*, *read_link*, *truncate_f*, *ftruncate_f*, *remove_f*, *utime_f*, *write_f*

fstat_vfs

Obtains statistics about mounted volume specified by file identifier.

```
#include <phile.h>
unsigned long fstat_vfs(
   unsigned long fid,    /* file identifier */
   struct statvfs *buf   /* volume statistics */
   )
```


Volume Types

All volumes.

Description

`fstat_vfs()` functions like `stat_vfs()` except that `fstat_vfs()` obtains the statistics about a volume by using the file identifier, not the pathname. The file identifier is first obtained with either an `open_f()` or an `open_fn()` call to any file in the volume.

Arguments

`file` Specifies the file identifier of the file (any file within the mounted volume).

`buf` Points to a `statvfs` structure defined in `<phile.h>`, as follows:

```
typedef struct {
   long val[2];
   } fsid_t;

struct statvfs {
   unsigned long f_bsize;     /* preferred volume block size */
   unsigned long f_frsize;    /* fundamental volume block size */
   unsigned long f_blocks;    /* total number of blocks */
   unsigned long f_bfree;     /* total number of free blocks */
   unsigned long f_bavail;    /* free blocks available to
                               * non-superuser */
   unsigned long f_files;     /* total # of file nodes
                               * (pHILE+ files only) */
   unsigned long f_ffree;     /* reserved (not supported) */
   unsigned long f_favail;    /* reserved (not supported) */
   fsid_t f_fsid;             /* reserved (not supported) */
   char f_basetype[16];       /* file system type name, null
                                 terminated */
   unsigned long f_flag;      /* reserved (not supported) */
   unsigned long f_namemax;   /* maximum filename length */
   char f_fstr[32];           /* file system specific string */
   unsigned long f_fstype;    /* file system type number */
   unsigned long f_filler[15];/* reserved (not supported) */
   };
```

Currently, the fields `f_ffree`, `f_favail`, `f_fsid`, `f_flag`, and `f_filler` are reserved and do not have values. For all volumes except pHILE+ format, the field `f_files` is unused.

The field `f_fstype` identifies the type of file system format. The values in `<phile.h>` are given below:

`FSTYPE_PHILE`	pHILE+ format volume
`FSTYPE_PCDOS`	MS-DOS format volume
`FSTYPE_CDROM`	CD-ROM format volume
`FSTYPE_NFS`	Client NFS volume

The return value for all unsupported fields is 0.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

`stat_vfs`

ftruncate_f

Changes the size of a file specified by its file identifier.

```
#include <phile.h>
unsigned long ftruncate_f(
   unsigned long fid,     /* file identifier */
   unsigned long length   /* file size in bytes */
   )
```


Volume Types

pHILE+, MS-DOS, and NFS formatted volumes.

Description

`ftruncate_f()` functions the same as `truncate_f()` except that `ftruncate_f()` changes the size of a file by using the file identifier instead of the pathname. The file identifier is first obtained with either `open_f()` or `open_fn()`. Unlike `annex_f()`, this system call changes both the logical and the physical file size. (`annex_f()` changes only the physical file size.)

On pHILE+ or MS-DOS volumes, the file must have been opened only once, that is no other task has it open and the calling task has opened it only once. If this is violated, the error `E_FOPEN` is returned.

On pHILE+ or MS-DOS volumes, if the file is truncated shorter than its `L_ptr`, the `L_ptr` is changed to the new end-of-file.

Arguments

`fid`	Specifies the file identifier associated with the file.
`length`	Specifies the new file size. If the file was previously longer than `length`, the extra bytes are truncated. If it was shorter, the bytes between the old and new lengths are filled with 0's.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

truncate_f, *open_f*, *open_fn*

get_fn Obtains the number of a file.

```
#include <phile.h>
unsigned long get_fn(
   char *name,          /* filename */
   unsigned long *fn    /* file number */
   )
```


Volume Types

pHILE+, MS-DOS, and CD-ROM formatted volumes.

Description

`get_fn()` returns the file number associated with a file. The file number can then be used with an `open_fn()` call or as part of an absolute filename.

Arguments

`name`	Points to the null-terminated pathname of the file.
`fn`	Points to the variable where `get_fn()` stores the file number.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

One usage of `get_fn()` is to get the current directory. In the example below, the current directory's FN is returned in `dir_fn`.

```
unsigned long dir_fn;         /*current directory */
if (get_fn(“.”, &dir_fn) != 0) {
   /*Insert some error processing here */;
}
```

The pseudocode below shows how to get the current directory's full pathname, rather than just the FN.

1. Get the directory's FN in `dir_fn` with `get_fn(".")` as above.
2. Open the parent directory ("..") with `open_dir()`.
3. Search the parent directory for the directory entry of the current directory.
   a. Read a directory entry with `read_dir()`.
   b. Compare the directory entry's `d_filno` with the FN of the current directory.
   c. Repeat steps a and b until they match.
   d. Remember the matching directory entry's `d_name`. It is the last component of the current directory's pathname.
4. Close the open directory with `close_dir()`.
5. Repeat steps 1-4 for the parent directory of the current directory, the grandparent of the current directory, etc., until reaching the root directory. The root directory is reached when either `get_fn()` of the parent directory is an error, or `get_fn()` of the parent directory is the same as `get_fn()` of the directory.
6. The answer is the concatenation of all the components found in step 3 d from last to first.

As stated above, `get_fn()` is available for local volumes only (not NFS volumes.)

To obtain not only the current directory, but also the current device, see `stat_f()`.

See Also

open_fn, *stat_f*

init_vol

Initializes a pHILE+ formatted volume.

```
#include <phile.h>
unsigned long init_vol(
   char *device,                          /* volume name */
   struct INIT_VOL_PARAMS *params,        /* parameters */
   void *scratchbuf                       /* scratch buffer */
   )
```


Volume Types

pHILE+ formatted volumes.

Description

`init_vol()` initializes a pHILE+ formatted volume. It does a logical format of the volume, setting up the necessary control structures and other information needed by the pHILE+ file system manager for subsequent file operations on the volume.

All volume parameters, except block size, are user-supplied in the params argument of `init_vol()`. Block size is specified by `fc_logbsize` in the pHILE+ configuration table, so all volumes initialized by `init_vol()` have the same block size.

`init_vol()` can be used for either first time initialization or reinitialization of a volume. Reinitialization quickly deletes all data on the volume. However, if the volume was initialized with a different block size this changes the block size.

Arguments

`device` Points to the null-terminated volume name.

`params` Points to an instance of the `init_vol_params` structure, which contains parameters used to initialize the volume. This structure is defined in `<phile.h>` as follows:

```
typedef struct init_vol_params {
   char volume_label[12];   /* volume label */
   unsigned long volume_size;  /* number of blocks in volume */
   unsigned long num_of_file_descriptors;   /* number of
                                    * descriptors in FLIST */
   unsigned long starting_bitmap_block_number;   /* first BITMAP
                                                    * block */
   unsigned long start_data_block_number;   /* first data
                                             * block */

   }INIT_VOL_PARAMS;
```

The fields of the `init_vol_params` structure are described below:

`volume_label`	Contains a 12-byte volume label. The pHILE+ file system manager copies the label to the volume's ROOTBLOCK but does not use it. (The volume label is not the volume name. The volume name contains the volume's major and minor device numbers.)
`volume_size`	The number of blocks on the volume. For example, a value of 5000 indicates the volume contains blocks 0 - 4999.
`num_of_file_descriptors`	The number of file descriptors in the volume's FLIST. This is the number of files that can be created on the volume.
`starting_bitmap_block_number`	The starting block for the volume's BITMAP.
`start_data_block_number`	The starting block for the volume's data blocks. The pHILE+ file system manager requires this parameter to be a multiple of 8.
`scratchbuf`	Points to a buffer that is used temporarily by the pHILE+ file system manager during initialization. The scratch buffer must be the size of a pHILE+ block. The pHILE+ Configuration Table parameter `fc_logbsize` (in the `sys_conf.h` file) determines this block size.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. All data stored on the volume is lost by execution of this call.
2. The volume's media must have been properly hardware formatted before this call is executed.
3. A mounted volume cannot be initialized.

4. The pHILE+ file system manager stores the volume's label and time of initialization in the volume's rootblock, but it does not use this information. The user decides how to use this information, which can be examined by using `read_vol()` to read the rootblock (block 2) directly.

5. The starting block of the bitmap also determines the starting block of the `FLIST`, since the `FLIST` immediately follows the bitmap.

6. `init_vol()` can initialize an unpartitioned hard disk. If an IDE disk is larger than the maximum CHS partition table entry size, i.e. larger than 8.4 gigabytes (8,422,686,720), an unpartitioned disk can be used to create one pHILE+ volume that fills the whole disk.

7. Except for the case in note 6, a hard disk must be partitioned before use. Use the pSOSystem utility `apps/dskpart`, the MS DOS utility `FDISK`, or another comparable utility provided by the SCSI board manufacturer. After partitioning, each partition that will contain a pHILE+ format volume must be formatted. This is already done if the disk was partitioned by `apps/dskpart`. Otherwise, separately format each partition that will contain a pHILE+ format volume with `init_vol()`.

8. `init_vol()` can initialize any size RAM disk with a pHILE+ format volume. Alternately, a RAM disk can be initialized with a DOS FAT format volume. (See *pcinit_vol*.)

See Also

`mount_vol`, *`pcinit_vol`*

link_f

Creates a hard link between two files on the same volume.

```
unsigned long link_f(
   char *name1,   /* an existing filename */
   char *name2    /* a new directory entry to be created */
   )
```

Volume Types

NFS formatted volumes.

Description

`link_f()` makes a hard link from `name2` to `name1`. This increments the link count for the file (see *stat_f*). After this call, `name1` and `name2` are two alternate names for the same file. Both files must be on the same volume.

Arguments

`name1`	Points to a null-terminated pathname of an existing file. Must not refer to a directory.
`name2`	Points to a null-terminated pathname of a directory entry to be created.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

symlink_f, *remove_f*

lock_f

Locks or unlocks part or all of an open file.

```
#include <phile.h>
unsigned long lock_f(
   unsigned long fid,        /* file identifier */
   unsigned long startpos,   /* starting lock position */
   unsigned long bcount      /* number of bytes to lock */
   )
```


Volume Types

pHILE+ formatted volumes.

Description

`lock_f()` locks or unlocks part or all of an open file. Following a `lock_f()` system call, only the task that set the lock can access the locked bytes and only through the connection (the file identifier) used to set the lock.

A `lock_f()` call replaces any previous locks it set through the same connection with the new lock. Thus, only one lock per connection can be set. `lock_f()` with `bcount` = 0 is used to remove a lock.

A file can have as many locks as it has connections if the locks do not overlap. If a task attempts to lock a region already locked through a different connection, an error is returned, even if the two connections are from the same task.

Arguments

`fid`	Specifies the file identifier of the file to lock.
`startpos`	Specifies the starting byte of the locked region.
`bcount`	Specifies the length of the locked region in bytes.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `lock_f()` enables the locked region to begin and/or end beyond the current logical or physical end of the file. In such cases, new data that is appended to the file in the locked region becomes locked.
2. `lock_f()` does not move the `L_ptr`.
3. When initially opened, a file connection has no locks.
4. When a connection to a file is closed, any lock it has on the file is removed.
5. A locked region of a file denies read, write, and truncate access to it by any other file connection. However, `annex_f()`, which expands a file's physical size without changing its logical size, is allowed.
6. Directory and system files cannot be locked.

See Also

annex_f

lseek_f

Repositions for read or write within an open file.

```
#include <phile.h>
unsigned long lseek_f(
   unsigned long fid,         /* file identifier */
   unsigned long position,    /* relative seek vector */
   long offset,               /* offset */
   unsigned long *old_lptr    /* previous L_ptr */
   )
```


Volume Types

All volume types.

Description

`lseek_f()` repositions the `L_ptr` associated with an open file connection. Each file connection has its own `L_ptr`, and it points to the next byte to be read or written in the file. Repositioning can be specified relative to the beginning of the file, the current `L_ptr`, or the end of the file.

Arguments

`fid` Specifies the file identifier associated with the file.

`position` Defines how to reposition `L_ptr` and must have one of the following values:

Value	Meaning
`0`	Offset from beginning of file
`1`	Offset from current `L_ptr`
`2`	Offset from end of file

`offset` Specifies the number of bytes to move `L_ptr`. A negative `offset` moves `L_ptr` backwards.

`old_lptr` Points to the variable where `lseek_f()` stores the previous value of the `L_ptr`.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

`lseek_f()` can be used to determine the current logical size of a file, as in this example:

```
lseek_f(fid, 2, 0, &oldptr)
lseek_f(fid, 0, oldptr, &filesize)
```

The first call seeks to the end-of-file and saves the original position. The second call restores the original position and obtains the end-of-file position. The end-of-file position is also the file's logical size.

Notes

1. A separate `L_ptr` is associated with each file connection. `lseek_f()` affects only the `L_ptr` associated with the specified file descriptor (`fid`).
2. Because `L_ptr` is unsigned, positioning it before the start of the file results in a `seek past end-of-file` error.
3. Because `L_ptr` cannot be moved beyond the end of the file, it is not possible to create a file with holes in it.

See Also

`read_f`, *`write_f`*

lstat_f

Gets the status of a symbolically linked file.

```
#include <phile.h>
unsigned long lstat_f(
   char *name,        /* file pathname */
   struct stat *buf   /* file status */
   )
```


Volume Types

NFS volumes.

Description

`lstat_f()` is like `stat_f()` except when the named file is a symbolic link. For a symbolic link, `lstat_f()` returns information about the link file, and `stat_f()` returns information about the file to which the link refers.

Arguments

`name` Points to the null-terminated pathname of a file.

`buf` Points to a `stat` structure defined in `<phile.h>`, as follows:

```
struct stat {
   mode_t st_mode;   /* ownership/protection */
   ino_t st_ino;     /* file ID */
   dev_t st_dev;     /* dev ID where the volume resides */
   dev_t st_rdev;    /* dev ID for character or block
                      * special files only */
   nlink_t st_nlink; /* number of hard links to the file */
   uid_t st_uid;     /* user ID */
   gid_t st_gid;     /* group ID */
   off_t st_size;    /* total size of file, in bytes */
   time_t st_atime;  /* file last access time */
   time_t st_mtime;  /* file last modify time */
   time_t st_ctime;  /* file last status change time */
   long st_blksize;  /* optimal block size for I/O ops */
   long st_blocks;   /* file size in blocks */
   };
```

No time zone is associated with the time values.

`mode_t`, `ino_t`, `dev_t`, `nlink_t`, `uid_t`, `gid_t`, `off_t`, and `time_t` are defined as `unsigned long` in `<phile.h>`.

The status information word `st_mode` consists of the following bits:

```
S_IFMT      0170000     /* type of file */
S_IFIFO     0010000     /* fifo special */
S_IFCHR     0020000     /* character special */
S_IFDIR     0040000     /* directory */
S_IFBLK     0060000     /* block special */
S_IFREG     0100000     /* regular file */
S_IFLNK     0120000     /* symbolic link */
S_IFSOCK    0140000     /* socket */
S_ISUID     0004000     /* set user ID on execution */
S_ISGID     0002000     /* set group ID on execution */
S_ISVTX     0001000     /* save swapped text even after use */
S_IREAD     0000400     /* read permission, owner */
S_IWRITE    0000200     /* write permission, owner */
S_IEXEC     0000100     /* execute/search permission, owner */
S_IRGRP     0000040     /* read permission, group */
S_IWGRP     0000020     /* write permission, group */
S_IXGRP     0000010     /* execute/search permission, group */
S_IROTH     0000004     /* read permission, other */
S_IWOTH     0000002     /* write permission, other */
S_IXOTH     0000001     /* execute/search permission, other */
```

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

symlink_f, *stat_f*, *fstat_f*

make_dir

Creates a directory file.

```
#include <phile.h>
unsigned long make_dir(
   char *name,          /* directory pathname */
   unsigned long mode   /* access permissions */
   )
```


Volume Types

All volume types, except CD-ROM; however, `mode` is used only for NFS volumes.

Description

`make_dir()` creates a new directory file.

Arguments

`name` Points to the null-terminated pathname of the directory file to create.

`mode` For NFS volumes only, specifies the access modes associated with the file and is the result of an OR operation performed on the following constants (defined in `<phile.h>`):

Mnemonic	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_IRUSR	Read permission, owner.
S_IWUSR	Write permission, owner.
S_IXUSR	Execute/search permission, owner.
S_IRGRP	Read permission, group.
S_IWGRP	Write permission, group.
S_IXGRP	Execute/search permission, group.
S_IROTH	Read permission, other.
S_IWOTH	Write permission, other.
S_IXOTH	Execute/search permission, other.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If the specified filename already exists, the new file is not created. An existing file must first be deleted by using `remove_f()` before its name can be used for a new file.
2. `make_dir()` creates only directory files. `create_f()` creates an ordinary file.

See Also

`create_f`, *`remove_f`*, *`open_dir`*

mount_vol

Mounts a pHILE+ formatted volume.

```
#include <phile.h>
unsigned long mount_vol(
   char *device,              /* volume name */
   unsigned long sync_mode    /* synchronization mode */
   )
```


Volume Types

pHILE+ formatted volumes.

Description

`mount_vol()` mounts a pHILE+ formatted volume. A volume must be mounted before any file operations can be applied to it. Permanent volumes (on non-removable media) need mounting only once. Removable volumes can be mounted and unmounted as required.

Arguments

`device`	Points to the null-terminated name of the volume to be mounted.	
`sync_mode`	Specifies the volume's write synchronization attribute. The attribute is defined in `<phile.h>` and must be set to one of the following values.	
	`SM_IMMED_WRITE`	Immediate write synchronization mode
	`SM_DELAY_DATE`	Immediate write all, except for file dates.
	`SM_CONTROL_WRITE`	Control write synchronization mode
	`SM_DELAYED_WRITE`	Delay write synchronization mode
	`SM_READ_ONLY`	Read only synchronization mode

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `mount_vol()` proceeds as if the designated pSOS+ device were mountable. A device is mountable if it is a true storage device that has been initialized by `init_vol()`.

2. The number of volumes that can be mounted simultaneously in the system cannot exceed the pHILE+ Configuration Table parameter `fc_nmount`.

3. The pHILE+ file system manager operates without regard for volume ownership. Furthermore, any task can perform a `mount_vol()`, and a mounted device has no record of the task that mounted it. Therefore, a volume is not automatically unmounted when the task that mounted it is deleted. If these or any security measures need to be addressed, the user's own layer of software must do so.

4. Not all mounted pHILE+ formatted volumes need to have the same block size. With the exception of block size 2^8 = 256, pHILE+ formatted volumes can be mounted provided their block size does not exceed the size of the cache buffers. pHILE+ formatted volumes with block size 256 can be mounted only if `fc_logbsize` is 8. This is done since most disks are block size 2^9 and disk drivers will return an error if the pHILE+ volume block size is less than the disk block size.

   `mount_vol()` determines the volume block size by checking for the root block using block size `fc_logbsize`, and then, if that fails, block sizes 9 through 15. Note, volumes with block size `fc_logbsize` are mounted fastest. If it does not find the root block, it returns error `E_VALIEN`. If it finds the root block but the volume's block size is greater than the cache buffer size, it returns error `E_BUFSIZE`. Otherwise it mounts the volume. For example, if `fc_logbsize` is 10 and `SC_PHILE_CDROM` is YES the cache buffers will be adjusted to 2^11 = 2048 bytes for the CD-ROMs. `mount_vol()` tries the block sizes in the following order: 10, 9, 11, 12, 13, 14, and 15. pHILE+ format volumes with block sizes 9, 10, or 11 can be mounted. Block size 8 will return error `E_VALIEN`. Block sizes 12 through 15 will return error `E_BUFSIZE`.

See Also

`init_vol`, *`pcmount_vol`*, *`nfsmount_vol`*, *`cdmount_vol`*, *`unmount_vol`*

move_f

Moves (renames) a file.

```
#include <phile.h>
unsigned long move_f(
   char *oldname,   /* old pathname */
   char *newname    /* new pathname */
   )
```


Volume Types

All volume types, except CD-ROM; however, some behavioral differences exist and are described here.

Description

`move_f()` changes the pathname associated with a file.

With one exception, the pHILE+ file system manager can move both ordinary and directory files on all volume types. The exception is directory files on MS-DOS formatted volumes, which cannot be moved. When a directory is moved, the directory and all files in the directory's subtree are moved.

Conceptually, `move_f()` moves a file by changing control structures on the volume (but no actual movement of data ever occurs). Therefore, `oldname` and `newname` must be on the same volume.

Arguments

`oldname`	Points to the null-terminated old pathname.
`newname`	Points to the null-terminated new pathname.

If `oldname` and `newname` are in the same directory, `move_f()` simply renames the file. Otherwise, `move_f()` has the effect of moving the file to a different location within the volume's directory tree. `move_f()` does not change the size or contents of the file.

`move_f()` fails if `newname` already exists or if the move operation would create a non-tree directory organization (for example, when a directory file is moved to its own subtree.)

If `oldname` is open, the file can be moved on pHILE+ and NFS volumes. An open file cannot be moved on MS-DOS volumes. Furthermore, no files can be moved on CD-ROM volumes.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

An MS-DOS directory can be moved. However, if the default directory of any task is the directory being moved or a subdirectory of the default directory being moved, that task's current directory will become incorrect. To move a directory follow the procedure below:

1. Get the attributes of the file or directory being moved with `stat_f()`.
2. If moving a file, i.e. `S_ISREG(buf.st_mode)` where `buf` is the attributes returned by `stat_f()`, move the file with `move_f()` and return. Otherwise, continue with step 3.
3. Create the new directory with `make_dir()`.
4. Open the directory being moved with `open_dir()`.
5. Move the entire contents into the new directory.
   a. Read a directory entry with `read_dir()`.
   b. If the directory entry is neither "." nor ".." then recurse to move the directory entry to the new directory.
   c. Repeat steps a and b until `read_dir()` returns `E_EOF`.
6. Close the open directory with `close_dir()`.
7. Delete the now empty old directory with `remove_f()`.

See Also

`make_dir`

nfsmount_vol Mounts a remote file system.

```
#include <pna.h>                /* for htonl() */
#include <phile.h>
unsigned long nfsmount_vol(
   char *device,                /* volume name */
   NFSMOUNT_VOL_PARAMS *params  /* parameters */
   )
```


Volume Types

NFS volumes.

Description

`nfsmount_vol()` mounts an NFS volume. A volume must be mounted before any file operations can be conducted.

Arguments

`device` Points to a null-terminated name of the volume to be mounted. Unlike the `mount_vol()` system call, the volume name provided does not correspond to a true pSOS+ device but to a pseudo-device. A pseudo-device does not necessarily correspond to any real device or device driver in the pSOS+ system. Drivers for this device number may or may not exist. In either case, the pHILE+ file system manager does not call them while it is accessing the NFS volume.

`params` Points to an instance of the `nfsmount_vol_params` structure, which contains parameters used for volume mounting and is defined in `<phile.h>` as follows:

```
typedef struct nfsmount_vol_params {
   unsigned long ipaddr;     /* Internet address of NFS server
                              * NOTE: network byte order
                              * Use htonl() */
   char *pathname;           /* pathname of filesystem to
                              * mount */
   unsigned long flags;      /* reserved; set to 0*/
   unsigned long retries;    /* # of NFS retries (0 is no
                              * retries*/
   unsigned long timeo;      /* timeout for each try (1/10
                              * seconds) */
   unsigned long reserved[4];/* reserved; set to 0 */
   } NFSMOUNT_VOL_PARAMS;
```

The fields of `nfsmount_vol_params` are defined as follows:

`ipaddr`	The IP address of the NFS host that contains the file system to mount. Since this is in network byte order, it should be set as follows: `params->ipaddr = htonl(address)`
`pathname`	Points to the pathname of the filesystem to mount.
`flags`	Reserved for future use and must be 0.
`retries`	The number of retries if communication with the NFS server times out. This does not include the initial try. Zero means try one time and do not attempt a retry.
`timeo`	The time-out interval for communication with the NFS server in tenths of a second. For backwards compatibility, if `timeo` is zero it is changed to 30 (3 seconds) and retries is changed to 2. These are the parameters used by earlier versions of pHILE+ that did not allow specifying these parameters.
`reserved`	Reserved for future use and must be 0.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The major device number of the volume name can exceed the maximum allowed device number in the pSOS+ Configuration Table because the device is *virtual*. A virtual device does not correspond to any device driver.
2. The number of volumes that can be mounted simultaneously in the system cannot exceed the pHILE+ Configuration Table parameter `fc_nmount`.

See Also

`mount_vol`, *`pcmount_vol`*, *`cdmount_vol`*, *`unmount_vol`*

open_dir Opens a directory file.

```
#include <phile.h>
unsigned long open_dir(
   char *dirname,  /* name of the directory file */
   XDIR *dir       /* pointer to buffer to
                    * return directory handle*/
   )
```


Volume Types

All volume types.

Description

`open_dir()` opens a designated directory file.

Arguments

`dirname`	Points to a null-terminated pathname of a directory file.
`dir`	Points to an `XDIR` structure, which is defined in `<phile.h>`.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

`close_dir`, *`read_dir`*

open_f Opens a file.

```
#include <phile.h>
unsigned long open_f(
   unsigned long *fid,  /* file identifier */
   char *name,          /* pathname */
   unsigned long mode   /* unused; set to zero */
   )
```

Volume Types

All volume types.

Description

`open_f()` creates a connection between a file and the calling task and returns a file identifier. The file identifier is used in subsequent operations on the file. `open_f()` fails if the system is out of file control blocks or if the task is out of open file table entries.

`open_f()` does not check for a file type. It opens ordinary files, directory files, or system files. However, directory files and system files are read-only.

`open_f()` always positions the `L_ptr` at the first byte in the file.

For CD-ROM volumes, `open_f()` can be used to read the *primary volume descriptor*. See *Primary Volume Descriptor*, under *Notes*.

If the file does not exist, `open_f()` creates it if `MO_CREATE` is set, or returns an error if not.

Arguments

`fid`	Points to the variable where `open_f()` stores the file identifier.
`name`	Points to the null-terminated pathname of the file to open.
`mode`	Optional create flag, MO_CREATE, and only on NFS. File access mode if created. Otherwise, should be set to 0 for future compatibility. If the file is created on NFS, it has a file access mode set to `mode & ~MO_CREATE`.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Primary Volume Descriptor

As a special case on CD-ROM volumes, the filename _VOLUME.Y in the root directory is used to read the primary volume descriptor, which is the starting point for locating all information on the volume. When you read _VOLUME.Y, pHILE+ omits the fields from it that are unused by your processor and byte-swaps the remaining fields to the proper order for the processor. Therefore, the primary volume descriptor can be read into the structure type that follows.

```
/* CD-ROM Primary Volume Descriptor as read from _VOLUME.$Y$ */
/*************************************************************/

#define CDFS_NAMMAX   32   /* max node size (in bytes) */

/* CD File System Directory Record (internal format) */

typedef struct dir_cdfs {
   USHORT dr_reclen;             /* directory record length */
   USHORT dr_xarlen;             /* extended attribute record length */
   ULONG dr_extent;              /* number of first data block in file */
   ULONG dr_fsize;               /* byte size of file data space */
   ULONG dr_cdate;               /* date when created (pSOS+ format) */
   ULONG dr_ctime;               /* time when created (pSOS+ format) */
   USHORT dr_flags;              /* directory flags per iso_dirrec */
   USHORT dr_namlen;             /* byte length of name */
   char dr_name[CDFS_NAMMAX + 1] /*the name */
} dir_cdfs_t;

/* CD File System Volume Descriptor template returned to user upon read of */
/* "_VOLUME.$Y$" virtual file */

typedef struct desc_cdfs {
   UCHAR cd_type                  /* volume descriptor type */
   UCHAR cd_id[5+1];              /* standard identifier */
   UCHAR cd_vers;                 /* volume descriptor version */
   UCHAR cd_flags;                /* volume flags */
   UCHAR cd_sysid[32+1];          /* system identifier */
   UCHAR cd_volid[32+1];          /* volume identifier */
   ULONG cd_volsize;              /* volume space size */
   UCHAR cd_escseq[32];           /* escape sequences */
   USHORT cd_volsetsize;          /* volume set size */
   USHORT cd_volseqnum;           /* volume sequence number */
   USHORT cd_logblksize;          /* logical block size */
```

```
    ULONG cd_pathtabsize;          /* path table byte size */
    ULONG cd_pathtab;              /* path table logical block */
    ULONG cd_pathtabopt;           /* opt path table log block */
    struct dir_cdfs cd_rootdir;    /* root directory */
    UCHAR cd_volsetid[128+1];      /* volume set identifier */
    UCHAR cd_pubid[128+1];         /* publisher identifier */
    UCHAR cd_prepid[128+1];        /* data preparer identifier */
    UCHAR cd_appid[128+1];         /* application identifier */
    UCHAR cd_cpyrid[37+1];         /* copyright file identifier */
    UCHAR cd_absfid[37+1];         /* abstract file identifier */
    UCHAR cd_bibfid[37+1];         /* bibliographic identifier */
    ULONG cd_cdate;                /* volume create date (pSOS+ format) */
    ULONG cd_ctime;                /* volume create time (pSOS+ format) */
    ULONG cd_mdate;                /* modification time (pSOS+ format) */
    ULONG cd_xdate;                /* expiration date (pSOS+ format) */
    ULONG cd_xtime;                /* expiration time (pSOS+ format) */
    ULONG cd_edate;                /* effective date (pSOS+ format) */
    UCHAR cd_svers;                /* file structure version */
    UCHAR cd_appdata[512];         /* application private */
}desc_cdfs_t;
```

See Also

open_fn, *close_f*

open_fn

Opens a file by its file identifier.

```
#include <phile.h>
unsigned long open_fn(
   unsigned long *fid,   /* file identifier */
   char *device,         /* volume name */
   unsigned long fn,     /* file number */
   unsigned long mode    /* unused, set to 0 */
   )
```


Volume Types

pHILE+, MS-DOS, and CD-ROM formatted volumes.

Description

`open_fn()` functions identically to `open_f()` except that `open_fn()` opens a file associated with a specified file number. The file number is first obtained with `get_fn()`.

`open_fn()` is more efficient than `open_f()` when a particular file is frequently opened, since `open_fn()` skips pathname parsing and directory searching.

Arguments

`fid`	Points to the variable where `open_fn()` stores the file identifier.
`device`	Points to the null-terminated name of the volume containing the file.
`fn`	The file number of the file.
`mode`	Optional create flag, MO_CREATE, and only on NFS. File access mode if created. Otherwise, should be set to 0 for future compatibility. If the file is created on NFS, it has a file access mode set to `mode & ~MO_CREATE`.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Primary Volume Descriptor

As a special case on CD-ROM volumes, the filename `_VOLUME.$Y$` in the root directory is used to read the primary volume descriptor. Refer to the description of `open_f()` on page 3-56 for details.

See Also

`open_f`, *`get_fn`*, *`close_f`*

pcinit_vol

Initializes an MS DOS FAT formatted volume.

```
#include <phile.h>
unsigned long pcinit_vol(
   char *device,        /* volume name */
   void *scratch_buf,   /* scratch buffer */
   unsigned long dktype /* type of volume */
   )
```


Volume Types

MS-DOS formatted volumes.

Description

`pcinit_vol()` initializes a volume in MS DOS format. `pcinit_vol()` performs a logical format of the volume, setting up the necessary control structures and other information needed by the pHILE+ file system manager for subsequent file operations on the volume. A volume must be initialized before it can be mounted.

Arguments

`device` Points to the null-terminated name of the volume to initialize.

`scratch_buf` Points to a 512-byte working buffer.

`dktype` Specifies the MS DOS media format and must have one of the following values:

Value	Mnemonic	Meaning
0	DK_HARD	Reinitialization of any volume, for example, a hard disk partition or any unpartitioned media format.
1	DK_360	360 Kbyte (5-1/4" double density)
2	DK_12	1.2 Mbyte (5-1/4" high density)
3	DK_720	720 Kbyte (3-1/2" double density)
4	DK_144	1.44 Mbyte (3-1/2" high density)
5	DK_288	2.88 Mbyte (3-1/2" high density)
6	DK_NEC	1.2 Mbyte (5-1/4" NEC)

7	DK_OPT	Optical disks, 124.4 Mbyte (Fuji M2511A OMEM)
8	DK_DRIVER	Initialization or reinitialization of any volume. Gets the media format from the disk driver. This requires pHILE+ 4.x.x disk driver support.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `pcinit_vol()` can be used for either first time initialization or reinitialization of a volume. However, pHILE+ 4.x.x disk driver support is required for first time initialization of hard disk partitions and unpartitioned disks other than the seven built-in media formats. Reinitialization never requires any disk driver support. Reinitialization quickly deletes all data on the volume.

2. There is an alternate interface to `pcinit_vol()`: `control_vol()` `cmd` `CONTROL_VOL_PCINIT_VOL`. (See *control_vol*.) It works the same as `pcinit_vol()` `dktype` `DK_DRIVER` except the format parameters are supplied in a parameter.

3. The different ways to do initialization and reinitialization of a DOS FAT format volume are listed below.

   - Reinitialization of any volume, for example, a partition or an unpartitioned disk: `pcinit_vol()` `dktype` `DK_HARD`.
   - Initialization of one of the seven built-in media types: `pcinit_vol()` `dktypes` `DK_360` through `DK_OPT`.
   - Initialization of any volume: `pcinit_vol()` `dktype` `DK_DRIVER`. This requires disk driver support. One example is initializing any magnetic optical disk except the one built-in type Fuji M2511A 124.4M optical disk.

- Initialization of any volume without disk driver support for DK_DRIVER: control_vol() cmd CONTROL_VOL_PCINIT_VOL. However, the application has to determine some of the format parameters.
- Initialization of any volume with nonstandard format parameters: control_vol() cmd CONTROL_VOL_PCINIT_VOL. One example is initializing a 1.44M 3 1/2" floppy disk in 720K format instead of the standard 1.44M format.

4. If the driver supports DK_DRIVER, it is not necessary to use any other dktype. DK_DRIVER can always be used. This has the advantage that the application can initialize a hard disk partition or any unpartitioned media type in the drive without change.
5. All data stored on the volume is lost by execution of this call.
6. The volume's hardware media must have been formatted before this call is executed.
7. A mounted volume cannot be initialized.
8. pcinit_vol() dktype DK_DRIVER can even be used to initialize an unpartitioned hard disk. However, this is rarely done. MS-DOS and Windows do not support unpartitioned hard disks, only partitioned hard disks. Thus an unpartitioned hard disk will not be interchangeable with MS-DOS and Windows.
9. Except for the rare case mentioned in Note 8, a hard disk must be partitioned before use. Use the pSOSystem utility apps/dskpart, the MS-DOS utility FDISK, or another comparable utility provided by the SCSI board manufacturer. After partitioning, each partition that will contain a DOS FAT format volume must be formatted. This is already done by the partitioning utility if the disk was partitioned by apps/dskpart or by most partition utilities provided by SCSI board manufactures. Otherwise, separately format each partition that will contain a DOS FAT format volume with pcinit_vol() dktype DK_DRIVER, if driver support is available, or with the MS-DOS FORMAT utility.
10. Using pcinit_vol() dktype DK_DRIVER any size RAM disk can be initialized with a DOS FAT format volume. Before DK_DRIVER a RAM disk had to be the size of one of the seven built-in types in order to be initialized with a DOS FAT format volume. Thus before DK_DRIVER pHILE+ format was usually used for RAM disks. (See *init_vol*.) Now with DK_DRIVER for any size RAM disk you can choose whether to use the RAM disk as DOS FAT format or pHILE+ format.

See Also

pcmount_vol, *init_vol*, *control_vol*() cmd CONTROL_VOL_PCINIT_VOL

pcmount_vol

Mounts an MS-DOS FAT formatted volume.

```
#include <phile.h>
unsigned long pcmount_vol(
   char *device,              /* volume name */
   unsigned long sync_mode    /* synchronization mode */
   )
```


Volume Types

MS-DOS formatted volumes.

Description

`pcmount_vol()` mounts an MS-DOS volume. A volume must be mounted before file operations can be applied to it.

Permanent (non-removable media) volumes need mounting only once. Removable volumes can be mounted and unmounted as required.

Arguments

`device`	Points to the null-terminated name of the volume to be mounted.	
`sync_mode`	Specifies the volume's write synchronization attribute. This attribute is defined in `<phile.h>` and must be set to one of the following values:	
	`SM_IMMED_WRITE`	Immediate write synchronization mode
	`SM_DELAY_DATE`	Immediate write all except file dates
	`SM_CONTROL_WRITE`	Control write synchronization mode
	`SM_DELAYED_WRITE`	Delay write synchronization mode
	`SM_READ_ONLY`	Read only synchronization mode

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. `pcmount_vol()` proceeds as if the designated pSOS+ device is mountable. A device is mountable if it has been initialized by `pcinit_vol()` or by the MS-DOS `FORMAT` command.
2. The number of volumes that can be mounted simultaneously in the system cannot exceed the pHILE+ Configuration Table parameter `fc_nmount` (from `sys_conf.h`).
3. The pHILE+ file system manager does not attempt verification or any other way of determining volume ownership. Any task can perform a `pcmount_vol()`. A mounted device does not retain a record of the task that mounted it. Therefore, a volume is not automatically unmounted when the task that mounted it is deleted. This and any other security measures, if desired, should be supported by the user's own layer of software.
4. For an application to mount an MS-DOS volume, the mount flag `FC_MSDOS` in `sys_conf.h` must be set.

See Also

`pcinit_vol`, *`mount_vol`*, *`nfsmount_vol`*, *`cdmount_vol`*, *`unmount_vol`*

read_dir

Reads directory entries in a file system independent format.

```
#include <phile.h>
unsigned long read_dir(
   XDIR *dir,         /* a directory handle */
   struct dirent *buf /* user structure to hold returned contents */
   )
```


Volume Types

All volume types.

Description

`read_dir()` reads one directory entry at a time from a directory file in a file system-independent format. The directory handle is first obtained with `open_dir()`.

Arguments

`dir` Points to the handle for the directory file, which has been returned by `open_dir()`.

`buf` Points to the memory area that receives the data. The data returned in `*buf` is a `dirent` structure defined in `<phile.h>`, as follows:

```
struct dirent {
   unsigned long d_filno;
   char d_name [MAXNAMLEN+1];
   }
```

`d_fileno` contains a number that is unique for each distinct file in the file system, and `d_name` contains a null-terminated filename, where the size is in the range of 1 through `MAXNAMLEN`+1. `MAXNAMLEN` is set to 255.

When the last entry has been read, an end-of-file error is returned.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Primary Volume Descriptor

As a special case on CD-ROM volumes, the filename `_VOLUME.$Y$` in the root directory is used to read the primary volume descriptor. Therefore, `_VOLUME.$Y$` is returned as one of the entries of the root directory. Refer to the description of `open_f()` on page 3-56 for details.

See Also

`open_dir`, *`close_dir`*

read_f Reads from a file.

```
#include <phile.h>
unsigned long read_f(
   unsigned long fid,      /* file identifier */
   void *buffer,           /* input buffer */
   unsigned long bcount,   /* byte read count */
   unsigned long *tcount   /* read count status */
   )
```


Volume Types

All volume types.

Description

`read_f()` reads data from a file, beginning at the current position of the connection's `L_ptr`.

After `read_f()`, the file's `L_ptr` is updated to point to the byte after the last byte that was read.

Arguments

`fid`	Specifies the file identifier associated with the file.
`buffer`	Points to the memory area to receive the data.
`bcount`	Specifies the number of bytes to read.
`tcount`	Points to the variable where `read_f()` stores the number of bytes actually read. The `tcount` value equals `bcount` unless the end-of-file was reached or an error occurred.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. On pHILE+, CD-ROM, and MS-DOS formatted volumes, `read_f()` operations are more efficient if `bcount` equals an integral multiple of the block size and the `L_ptr` is positioned at a block boundary.
2. On pHILE+, CD-ROM, and MS-DOS formatted volumes, if the requested data includes entire blocks or a contiguous sequence of blocks and if such blocks are not already in the buffer cache, the pHILE+ file system manager reads these blocks directly into the caller's buffer (without going through the buffer cache).
3. `read_f()` automatically positions the `L_ptr` for sequential read operations. If random reads are necessary, use `lseek_f()` to reposition the `L_ptr`.

See Also

`lseek_f`, *`read_vol`*

read_link

Reads the value of a symbolic link.

```
#include <phile.h>
unsigned long read_link(
   char *name,              /* a file containing the symbolic link */
   char *buf,               /* user buffer to hold the returned contents */
   unsigned long *bufsize   /* maximum buffer size */
   )
```


Volume Types

NFS volumes.

Description

`read_link()` reads the contents of the symbolic link of a file. The returned data is *not null-terminated*.

Arguments

`name`	Points to the null-terminated pathname of the file containing the symbolic link.
`buf`	Points to the memory area that receives the data.
`bufsize`	Points to the maximum buffer size before the call, and the length of the data returned in `buf` after the call.

If successful, `read_link` stores in `*bufsize` the length of the data stored in `buf`. If this is the same as the maximum buffer size, only part of the data may have been returned.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

The following example is a typical call to `read_link()` with all the code necessary for full error checking.

```
#define MAX_RESULT 100          /* Use 1 more than the longest
                                 * expected result. */
{
   char contents[MAX_RESULT+1]; /* Contents of symbolic link */
   unsigned long size;         /* Size of contents */

   size = MAX_RESULT;          /* Room available in contents */

   if (read_link("3.2/sym_link", &contents[0], &size) != 0)
      /* Error processing for failed system call */;

   contents[size] = '\0';      /* Null terminate the result */

   if (size == MAX_RESULT)       /* Possible partial result */
      /* Error processing for possible partial result */;
}
```

See Also

lstat_f, *symlink_f*

read_vol

Reads directly from a pHILE+ formatted volume.

```
#include <phile.h>
unsigned long read_vol(
   char *device,          /* volume name */
   unsigned long block,   /* base block */
   unsigned long index,   /* byte offset */
   unsigned long bcount,  /* number of bytes to read */
   void *buffer           /* input buffer */
   )
```


Volume Types

pHILE+, MS-DOS, and CD-ROM formatted volumes.

Description

`read_vol()` reads data directly from a volume, bypassing the file system organization imposed by the pHILE+ file system manager.

Arguments

`device`	Points to the null-terminated name of the volume to read.
`block`	Identifies the logical block number to begin reading.
`index`	Specifies where to begin reading within the specified block.
`bcount`	Specifies the number of bytes to read.
`buffer`	Points to the memory area to receive the data.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If `index` is larger than the volume's block size, the read begins in a subsequent block. For example, on a volume with a 1024-byte block size, a read of block 5, index 1224, is the same as a read block 6, index 200.
2. CD-ROM volumes generally use a 2K block size.
3. `read_vol()` does not check for the end of the volume, so blocks beyond the specified volume size can be read if they physically exist.
4. If the requested data includes either entire blocks or a contiguous sequence of blocks and if such blocks are not already in the buffer cache, the pHILE+ file system manager reads blocks directly into the buffer (without going through the buffer cache). Therefore, `read_vol()` executes more efficiently when `bcount` and `index` are equal to integral multiples of blocks.

See Also

`write_vol`

remove_f Deletes a file.

```
#include <phile.h>
unsigned long remove_f(
   char *name   /* pathname */
   )
```


Volume Types

All volume types, except CD-ROM; however, functional differences exist and are described here.

Description

`remove_f()` deletes a file from a volume. The file can be an ordinary file or a directory file. All storage used by the file is returned to the system for reuse. The file's entry in its parent directory is also deleted.

System files and non-empty directory files cannot be deleted. On pHILE+ and MS-DOS formatted volumes, an open file cannot be deleted. An open file can be deleted on an NFS volume. CD-ROM volumes are read-only.

Arguments

`name`	Points to the null-terminated pathname of the file to delete.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

`create_f`, *`make_dir`*

stat_f

Gets the status of a named file.

```
#include <phile.h>
unsigned long stat_f(
   char *name,         /* file pathname */
   struct stat *buf    /* file status */
   )
```

Volume Types

All volumes.

Description

`stat_f()` returns information about the named file. This call does not need read, write, or execute permission of the named file. It does need execute/search permission of all the directories leading to the named file.

NOTE: When you use `stat_f` on a file under NFS, the mtime field and other time-related fields do not reflect the correct times. This is a limitation of `stat_f`.

Arguments

`name` Points to the null-terminated pathname of the file.

`buf` Points to a `stat` structure defined in `<phile.h>` as follows:

```
struct stat {
   mode_t st_mode;   /* ownership/protection */
   ino_t st_ino;     /* file ID */
   dev_t st_dev;     /* device ID where the volume resides */
   dev_t st_rdev;    /* device ID, for character or block
                      * special files only */
   nlink_t st_nlink; /* number of hard links to the file */
   uid_t st_uid;     /* user ID */
   gid_t st_gid;     /* group ID */
   off_t st_size;    /* total size of file, in bytes */
   time_t st_atime;  /* file last access time */
   time_t st_mtime;  /* file last modify time */
   time_t st_ctime;  /* file last status change time */
   long st_blksize;  /* optimal block size for I/O ops */
   long st_blocks;   /* file size in blocks */
   };
```

mode_t, ino_t, dev_t, nlink_t, uid_t, gid_t, off_t, and time_t are defined as unsigned long in <phile.h>. The following differences exist for local file systems (pHILE+, MS-DOS, and CD-ROM):

rdev = dev, nlink = 1, uid = 0, gid = 0, atime = ctime = mtime

The status information word st_mode contains the following bits:

```
S_IFMT      0170000     /* type of file */
S_IFIFO     0010000     /* fifo special */
S_IFCHR     0020000     /* character special */
S_IFDIR     0040000     /* directory */
S_IFBLK     0060000     /* block special */
S_IFREG     0100000     /* regular file */
S_IFLNK     0120000     /* symbolic link */
S_IFSOCK    0140000     /* socket */
S_ISUID     0004000     /* set user ID on execution */
S_ISGID     0002000     /* set group ID on execution */
S_ISVTX     0001000     /* save swapped text even after use */
S_IRUSR     0000400     /* read permission, owner */
S_IWUSR     0000200     /* write permission, owner */
S_IXUSR     0000100     /* execute/search permission, owner */
S_IRGRP     0000040     /* read permission, group */
S_IWGRP     0000020     /* write permission, group */
S_IXGRP     0000010     /* execute/search permission, group */
S_IROTH     0000004     /* read permission, other */
S_IWOTH     0000002     /* write permission, other */
S_IXOTH     0000001     /* execute/search permission, other */
```

The time_t quantities are the number of seconds since 00:00:00 January 1, 1970. These times can be converted using the following standard ANSI C functions available in pREPC+:

`struct tm *localtime(const time_t *tp)`	Converts `time_t` into struct tm which contains month, day, year, hour, minute, second.
`char *asctime(const struct tm *tp)` or `asctime_r()`.	Converts `struct tm` into a string.
`char *ctime(const time_t *tp)` or `ctime_r()`	Converts `time_t` into a string.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

`stat_f()` can be used to determine both the current device and the current directory of local volumes. Thus, you can use `stat_f()` to return to a device and directory after leaving them, or to construct absolute path names starting at the current directory. Only the directory file number is available, not the full directory path. To obtain the full directory path, see `get_fn()`.

```
/* Obtaining both the current device and the current directory */

ULONG rc;                     /* System call return code */
struct stat   current_stat;   /* stat_f() of "." */
ULONG device;                 /* Current device */
ULONG directory;             /* Current directory */

if((rc = stat_f(".", &current_stat)) != 0)
   /* Error processing */

device = current_stat.st_dev;
directory = current_stat.st_ino;

/* Returning to the above device and directory at a later time. */
```

```
char dirpath[30];      /* To change back */

sprintf(dirpath, "0x%04x.0x%02x.0x%02x.0x%08x/.",
   device >> 16,   /* Major device number */
   (device >> 8) & 0xffU,   /* Minor device number */
   device & 0xffU,   /* Partition number */
   directory);   /* File number to start at */

if((rc = change_dir(dirpath)) != 0)
   /* Error processing */

/* Constructing absolute path name starting at the saved directory */

#define REL_PATH_LEN 8  /* Length of path below saved directory */
char path[29 + REL_PATH_LEN];      /* Absolute path of file.txt */

sprintf(path, "0x%04x.0x%02x.0x%02x.0x%08x/%s",
   device >> 16,            /* Major device number */
   (device >> 8) & 0xffU,   /* Minor device number */
   device & 0xffU,          /* Partition number */
   directory,               /* File number to start at */
   "file.txt");   /* Relative path below saved directory */
```

See Also

fstat_f, *chmod_f*, *fchmod_f*, *chown_f*, *fchown_f*, *link_f*, *read_f*, *read_link*, *truncate_f*, *ftruncate_f*, *remove_f*, *utime_f*, *write_f*

stat_vfs

Gets statistics for a named volume.

```
#include <phile.h>
unsigned long stat_vfs(
   char *name,           /* file pathname */
   struct statvfs *buf   /* volume statistics */
   )
```

Volume Types

All volume types.

Description

`stat_vfs()` returns information about a mounted volume.

Arguments

`name` Points to a null-terminated pathname of any file within the mounted volume.

`buf` Points to a `statvfs` structure defined in `<phile.h>`, as follows:

```
typedef struct {
   long val[2];
   } fsid_t;

struct statvfs {
   unsigned long f_bsize;     /* preferred volume block size */
   unsigned long f_frsize;    /* fundamental volume block size */
   unsigned long f_blocks;    /* total number of blocks */
   unsigned long f_bfree;     /* total number of free blocks */
   unsigned long f_bavail;    /* free blocks available to
                               * non-superuser */
   unsigned long f_files;     /* total # of file nodes
                               * (pHILE+ files only) */
   unsigned long f_ffree;     /* reserved (not supported) */
   unsigned long f_favail;    /* reserved (not supported) */
   fsid_t f_fsid;             /* reserved (not supported) */
   char f_basetype[16];       /* file system type name, null
                                 terminated */
   unsigned long f_flag;      /* reserved (not supported) */
   unsigned long f_namemax;   /* maximum filename length */
   char f_fstr[32];           /* file system specific string */
   unsigned long f_fstype;    /* file system type number */
   unsigned long f_filler[15];/* reserved (not supported) */
   };
```

Currently, the fields f_ffree, f_favail, f_fsid, f_flag, and f_filler are reserved and do not have values. For all volumes except pHILE+ format, the field f_files is unused.

The field f_fstype identifies the type of file system format. The values in <phile.h> are given below:

FSTYPE_PHILE	pHILE+ format volume
FSTYPE_PCDOS	MS-DOS format volume
FSTYPE_CDROM	CD-ROM format volume
FSTYPE_NFS	Client NFS volume

The return value for all unsupported fields is 0.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Fields f_bsize and f_frsize are the preferred file system block size, and the fundamental file system block size. The value of f_bsize is always a multiple of f_frsize. For best efficiency transfers should be an even multiple of f_bsize bytes and begin on a f_bsize boundary. If that is not possible, transfers should be a multiple of f_frsize and begin on a f_frsize boundary to avoid blocking/deblocking. On a MS-DOS FAT format volume f_bsize is the cluster size in bytes, and f_frsize is the block size, i.e. 512. On a NFS client volume f_bsize is the optimal transfer size, and f_frsize is the block size, both from the STATFS protocol. On a CD-ROM format volume both f_bsize and f_frsize are 2,048. On a pHILE+ format volume both f_bsize and f_frsize are the volume's block size which is between 2^8 and 2^15.

2. The field f_basetype[] names the type of the file system format. The values are given below:

Value	Description
CD-ROM	CD-ROM format volume
PC/DOS	MS-DOS FAT format volume
NFS	NFS client volume
pHILE+	pHILE+ format volume

3. The field `f_namemax` is the maximum length filename supported by the filesystem. In the case of NFS client the value is due to only local pHILE+ NFS code. If the remote NFS server has a smaller limit, that smaller limit will prevail.

4. The field `f_fstr[]` is file system specific. On pHILE+ format it is the volume label when the volume was initialized by `init_vol()`, for example: `struct init_vol_params.volume_label[]`. On MS-DOS FAT format it is the volume label from the boot record, not from the root directory Volume attribute file entry. On CD-ROM it is the volume identifier from the primary volume descriptor, for example: `struct desc_cdfs.cd_volid[]`. On NFS client it is not used.

See Also

fstat_vfs

symlink_f

Creates a symbolic link to a file.

```
unsigned long symlink_f(
   char *name1,   /* a string used in creating the
                   * symbolic link */
   char *name2    /* the name of the file to be created */
   )
```


Volume Types

NFS volumes.

Description

`symlink_f()` creates a symbolic link `name1` in the file `name2`. The files do not need to be on the same volume.

The file to which the symbolic link points is used when an `open_f()` is performed on the link. A `stat_f()` performed on a symbolic link returns the linked-to file (whereas `lstat_f()` returns information about the link itself). `read_link()` can be used to read the contents of a symbolic link.

Arguments

`name1`	Points to the null-terminated pathname of the symbolic link.
`name2`	Points to the null-terminated pathname of the file.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

read_link, *link_f*, *remove_f*

sync_vol Synchronizes a volume.

```
#include <phile.h>
unsigned long sync_vol(
   char *device   /* volume name */
   )
```

Volume Types

pHILE+ and MS-DOS formatted volumes.

Description

`sync_vol()` updates a mounted volume by writing all modified volume information to the physical device. Updated files, descriptors, and all cache buffers that contain physical blocks are flushed to the device.

This call enables manual updating of a volume and is irrelevant in relation to immediate-write synchronization mode. CD-ROM volumes are read-only.

Arguments

`device`	Points to the null-terminated name of the volume to synchronize.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

Because no inherent access restrictions exist with respect to a volume, any task can call `sync_vol()`. `sync_vol()` keeps the volume busy during the update.

See Also

unmount_vol, *mount_vol*, *pcmount_vol*

truncate_f Changes the size of a named file.

```
#include <phile.h>
unsigned long truncate_f(
   char *name,            /* file pathname */
   unsigned long length   /* file size in bytes */
   )
```

Volume Types

pHILE+, MS-DOS, and NFS volumes.

Description

`truncate_f()` causes the file specified by `name` to have a size (in bytes) equal to `length`. If the file was previously longer than `length`, the extra bytes are truncated. If it was shorter, the bytes between the old and new lengths are filled with zeroes.

Unlike `annex_f()`, this system call changes both the logical and the physical file size. (`annex_f()` changes only the physical file size.)

On pHILE+ or MS-DOS volumes, the file must not be open. If this is violated, the error `E_FOPEN` is returned.

Arguments

`name`	Points to the null-terminated pathname of the file.
`length`	Specifies the new file size in bytes.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

ftruncate_f, *open_f*, *open_fn*

unmount_vol Unmounts a volume.

```
#include <phile.h>
unsigned long unmount_vol(
   char *device   /* volume name */
   )
```

Volume Types

All volume types.

Description

`unmount_vol()` unmounts a previously mounted volume. Unmounting a volume causes it to be synchronized. Synchronization causes all memory-resident volume data to be flushed to the device.

Arguments

`device`	Points to the null-terminated name of the volume to unmount.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Any task can unmount a volume. If some security is needed, the user must supply the bookkeeping software to keep track of volumes and tasks that perform the mounts.
2. Conceptually, unmounting a volume is unnecessary unless it is physically removed and a new volume is mounted on the same device. However, a limit exists to the number of volumes that can be mounted simultaneously, and unmounting frees entries in the volume mount table.
3. Once unmounted, a volume is inaccessible.

4. unmount_vol() fails and returns error E_MNTOPEN if any open files exist on the volume.
5. unmount_vol() fails if any unretried I/O error occurs while writing to disk any changed blocks cached in memory.
6. If due to I/O errors a disk can not be unmounted with unmount_vol(), it can be unmounted with control_vol() cmd CONTROL_VOL_CHANGED_VOL arg CONTROL_VOL_ALWAYS_CHANGED. However, any changed blocks cached in memory will be lost.

See Also

mount_vol, *pcmount_vol*, *nfsmount_vol*, *cdmount_vol*, *control_vol*() cmd CONTROL_VOL_CHANGED_VOL

utime_f

Sets the access and modification times of a file.

```
#include <phile.h>
unsigned long utime_f(
   char *name,           /* file pathname */
   struct utimbuf *times /* file access and modification times */
   )
```

Volume Types

NFS volumes.

Description

`utime_f()` sets the access and modification times of a file.

Arguments

`name` Points to the null-terminated pathname of the file.

`times` If `times` is NULL, the access and modification times are set to the current time. Otherwise, `times` is interpreted as a pointer to a `utimbuf` structure defined in `<phile.h>` as follows:

```
struct utimbuf {
   time_t actime;    /* access time */
   time_t modtime;   /* modification time */
   };
```

No time zone is associated with the time values.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

See Also

stat_f, *fstat_f*

verify_vol

Verifies a volume's control structures.

```
#include <phile.h>
unsigned long verify_vol(
   char *device,                /* volume name */
   VERIFY_VOL_PARAMS *params    /* parameters */
   )
```


Volume Types

pHILE+ formatted volumes.

Description

`verify_vol()` examines all control structures on a pHILE+ formatted volume. Inconsistencies are reported to a user-supplied callout routine, which can relay further instructions. The callout routine can then request `verify_vol()` to correct the inconsistency.

Usage instructions for `verify_vol` are provided in *Usage* on page 3-94.

Arguments

`device` Points to the null-terminated name of the volume to be verified.

`params` Points to an instance of the `verify_vol_params` structure defined in `<phile.h>` as follows:

```
typedef struct verify_vol_params {
   void *pb_dataptr;                /* work area pointer */
   unsigned long pb_datalen;        /* length of work area */
   unsigned long pb_maxdepth;       /* maximum depth of
                                     * directory tree */
   fault_desc_block *pb_fdbptr;     /* fault descriptor block
                                     * pointer */
   unsigned long (*pb_faultp)(void);/* faultp function */
   unsigned long *pb_badblkptr;     /* bad block list */
   } VERIFY_VOL_PARAMS;
```

The contents of the verify_vol_params fields are as follows:

pb_dataptr — Points to a work area required by verify_vol(). The size of this work area (in bytes) is given by the formula:

1072 + (n * *vsize*) + (260 * *nfd*) + (524 * *maxdepth*)

where *vsize* is the size of the volume in blocks; *nfd* is the number of file descriptors specified when the volume was initialized; n is determined from nfd in the table below:

Value of nfd	Value of n
1 ≤ nfd ≤ 2^8-4	1
2^8-3 ≤ nfd ≤ 2^16-4	2
2^16-3 ≤ nfd ≤ 2^24-4	3
2^24-3 ≤ nfd ≤ 2^28-4	4

and *maxdepth* is the maximum depth of the directory tree (it should be equal to pb_maxdepth, below.)

For example, on a volume with:

vsize = 5000 blocks
nfd = 100 entries
maxdepth = 5 levels

a total of 1072 + (1x5000) + (260x100) + (524x5) = 34,692 bytes would be required. This work area can be statically allocated, or it can be dynamically allocated by a pSOS+ rn_getseg() call.

pb_datalen — The size (in bytes) of the work area pointed to by pb_dataptr. The verify_vol() call uses this entry to confirm that the work area is large enough.

pb_maxdepth — The maximum depth of the volume's directory tree. If any branch of the directory exceeds this depth, verify_vol() terminates and returns an error code to the calling task. The minimum value allowed is 1, which indicates a flat directory (i.e., one containing no subdirectories).

`pb_fdbptr`	Points to a fault descriptor block (FDB) in the caller's memory area. When a fault is detected, `verify_vol()` places a detailed description of the fault into the FDB. The FDB format is described on page 3-97.
`pb_faultp`	Points to the user-provided `faultp()` procedure that is called each time `verify_vol()` detects a fault. `faultp()` is responsible for processing the fault. Refer to *faultp()* on page 3-95 for more details.
`pb_badblkptr`	Points to a user-provided list of *bad blocks* on the volume. A bad block is a block that cannot be read and/or written and is therefore unusable by the pHILE+ file system manager. This list is made up of 32-bit entries and is terminated with a 0 entry. The entries need not be in any specific order. `verify_vol()` can greatly simplify the handling of bad blocks. Refer to *Bad Blocks* on page 3-104 for information on this feature. If no bad block list is provided, this entry must be 0.

Target

`verify_vol()` calls a user-supplied function, `faultp()`, for status-checking (see page 3-95). For each processor family, `faultp()` returns its return value in the register specified below:

On 68K processors, `faultp()` uses the D0.L register.

On PowerPC processors, `faultp()` uses the r3 register.

On 960 processors, `faultp()` uses the g0 register.

On x86 processors, `faultp()` uses the %eax register.

On MIPS processors, `faultp()` uses the v0 register.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Usage

`verify_vol()` can be used to perform the following actions:

Volume Integrity Verification — `verify_vol()` examines all volume control structures to verify their consistency. Inconsistencies are reported and described in detail.

Volume Correction — Certain kinds of inconsistencies can be corrected.

Bad Block Elimination — Bad blocks can be marked as "in use" in the volume bitmap, thus excluding them from allocation by pHILE+ file system manager.

`verify_vol()` can be used in two ways. First, it can be used to perform a simple test of correctness, for example, at each power-on or system restart. Second, it can be integrated into a volume repair utility with a user-supplied interface.

Under normal operating conditions, pHILE+ file system manager always maintains the volume control structures in a correct and consistent state. `verify_vol()` is most useful when used following a system error or failure that can corrupt the file system, such as one of the following:

- A power failure, or a CPU or disk controller crash. In such cases, pHILE+ file system manager can be interrupted in the middle of a critical operation, resulting in a corrupted file system.
- A hard error or data corruption in one or more blocks containing volume control structures.
- Errors in the user-supplied physical disk driver.
- Restarting a task in pHILE+ file system manager.

Requirements and Restrictions

1. `verify_vol()` suspends all other I/O transactions to the designated volume. Because of the time required to execute it, `verify_vol()` should be called when the volume is idle.
2. Executing `verify_vol()` requires that the volume is mounted and no files are open.
3. `verify_vol()` cannot be used on an MS-DOS, CD-ROM, or NFS volume.

Functional Description

On the specified volume, `verify_vol()` examines the volume's control structures and searches for *faults*. A fault is any inconsistency in the control structures. For example, the volume's bitmap may indicate a particular block is free, while in fact it is being used. In all, there are 42 different kinds of faults detectable by `verify_vol()`.

`verify_vol()` examines the following volume control structures:

- Root block
- Bitmap
- FLIST
- All directories
- All file indirect blocks
- All file index blocks

`verify_vol()` stores a detailed description of the detected fault into a user-provided fault descriptor block (FDB) and then calls the user-provided function `faultp()`, described below.

faultp()

`faultp()` is called by `verify_vol()`without any parameters. `faultp()` is responsible for additional processing of the fault.

`verify_vol()` calls `faultp()`with the following information:

- The type of fault
- A detailed description of the fault

- An indication of whether or not the fault is correctable

`faultp()` performs its own check and returns a status code in the register supported by pHILE+, which is processor-specific. The register used on each processor family is specified in *Target* on page 3-93.

The status code `faultp()` returns must be one of the following:

Code	Description
`0`	Continue volume verification without correcting the fault.
`1`	Correct fault and continue volume verification.
`2`	Terminate volume verification.

If status = 1 is returned, `verify_vol()` makes modifications to the volume, which correct the fault. In most cases, the obvious modification is made. Less obvious modifications are described in *Bad Blocks* on page 3-104. Note that status = 1 (correct fault) should be returned only if the FDB indicates the fault is fixable (refer to the FDB description below). If status = 1 is returned for a non-fixable fault, it is ignored and verification continues.

If `verify_vol()` is being used simply to verify volume correctness, then, when called, `faultp()` can return a status of 0, which continues the rest of volume verification, or a status of 2, which terminates `verify_vol()` and returns an error to the caller.

`verify_vol()` can also be integrated into a “volume repair” utility with an operator interface. This utility should implement `faultp()` so that it will

- Display each fault in detail;
- Indicate if the fault is fixable;
- And if so, ask the user if he wants it fixed;
- And if so, return status = 1 to `verify_vol()`.

Faults that are not fixable may require additional user action. For example, you may perform the following steps repeatedly:

1. Use `verify_vol()` to correct all fixable errors and obtain a list of non-fixable errors.
2. Examine, copy, and delete the affected files, as required.

When step 1 produces no more faults, you can consider the volume corrected.

The Fault Descriptor Block (FDB)

The structure `fault_desc_block` defines the FDB in `phile.h` as follows:

```
typedef struct fault_desc_block {
   unsigned long fdb_code;   /* fault code */
   unsigned long fdb_fn1;    /* file number for file 1 */
   unsigned long fdb_fn2;    /* file number for file 2 */
   char *fdb_path1;          /* pathname for file 1 */
   char *fdb_path2;          /* pathname for file 2 */
   unsigned long fdb_bn;     /* block number */
   unsigned long fdb_fixable;/* fault fixable indicator */
   } FAULT_DESC_BLOCK;
```

The contents of the `fault_desc_block` fields are as follows:

`fdb_code` Contains a fault code describing the type of fault.

`fdb_fn1` Contains the file number of the file.

`fdb_fn2` For faults involving two files, contains the file number of the second file.

`fdb_path1` Contains a pointer to the file's complete pathname. This pathname is constructed by `verify_vol()` within the `verify_vol()` work area.

`fdb_path2` For faults involving two files, contains a pointer to the second file's complete pathname.

`fdb_bn` For faults involving a specific block, contains the block number of the affected block.

`fdb_fixable` Indicates whether the fault can be corrected by `verify_vol()`, as follows:

`fdb_fixable = 0` means fault is *not* fixable.

`fdb_fixable = 1` means fault is fixable.

Fault Types

Table 3-3 beginning on page 3-98 summarizes, for each fault type, the contents of each field. An X indicates the field is used in describing the fault. The last column indicates whether or not the fault is fixable.

NOTE: Footnotes *1* through *9* for Table 3-3 are all listed at the end of the table.

TABLE 3-3 Fault Summary

Mnemonic	Description	Hex	FN1	FN2	PATH 1	PATH 2	BN	Fix-able
VF_BMOFL	The bitmap and FLIST[1], as specified in ROOTBLOCK[2], overlap.	2101						N
VF_BMSIZ	The bitmap size and volume size, as specified in ROOTBLOCK, are inconsistent with one another.	2102						N
VF_FLSIZ	The FLIST size and number of file descriptors, as specified in ROOTBLOCK, are inconsistent with one another.	2103						N
VF_BMOVL	The bitmap, as specified in ROOTBLOCK, extends beyond the end of the volume.	2104						N
VF_FLOVL	The FLIST, as specified in ROOTBLOCK, extends beyond the end of the volume.	2105						N
VF_BMDA	The bitmap, as specified in ROOTBLOCK, overlaps the volume's data area.	2106						N
VF_FLDA	The FLIST, as specified in ROOTBLOCK, overlaps the volume's data area.	2107						N
VF_BMEXT	The extent map in the bitmap FD[3] disagrees with ROOTBLOCK.	2108						N
VF_FLEXT	The extent map in the FLIST FD disagrees with ROOTBLOCK.	2109						N
VF_NDRFD	The FD for the ROOT directory does not indicate it is a directory.	210A						Y

TABLE 3-3 Fault Summary (Continued)

Mnemonic	Description	Hex	FN1	FN2	PATH 1	PATH 2	BN	Fix-able
VF_FDMU	A FD is used by more than one file. `verify_vol()` returns the FN[4] and path-name of both files, although the FNs are the same[5].	210B	X	X	X	X		Y
VF_FDFRE	A FD that is in use is marked free.	210C	X		X			N
VF_FDUSE	A FD that is not in use is marked as in use.	210D	X					Y
VF_NSSFD	The FD of a non-system file indicates the file is a system file.	2110	X		X			Y
VF_SNSFD	The FD for a system file (`ROOTBLOCK`, `BITMAP`, or `FLIST`) does not indicate the file is a system file.	2111	X		X			Y
VF_PARFD	The parent FN within a FD does not point to the file's parent directory.	2112	X		X			Y
VF_FCFD	The file count within a FD for a directory is incorrect.	2113	X		X			Y
VF_SIZFD	A file's FD indicates that its logical size is greater than its physical size.	2114	X		X			Y
VF_ANXFD	A file has an annex size of 0 in its FD. This fault is corrected by setting the annex size to 1.	2115	X		X			Y
VF_EXTFD	See *Extent Map Faults* on page 3-103.	2118	X		X		X	N
VF_INFD	See *Extent Map Faults* on page 3-103.	2119	X		X		X	N

TABLE 3-3 Fault Summary (Continued)

Mnemonic	Description	Hex	FN1	FN2	PATH 1	PATH 2	BN	Fix-able
VF_IXFD	See *Extent Map Faults* on page 3-103.	211A	X		X		X	N
VF_TBCFD	See *Extent Map Faults* on page 3-103.	211B	X		X			Y
VF_LLBFD	See *Extent Map Faults* on page 3-103.	211C	X		X			Y
VF_LLBIN	See *Extent Map Faults* on page 3-103.	211D	X		X			Y
VF_EXTIN	See *Extent Map Faults* on page 3-103.	211E	X		X		X	N
VF_INIX	See *Extent Map Faults* on page 3-103.	211F	X		X		X	N
VF_LLBIX	See *Extent Map Faults* on page 3-103.	2120	X		X		X	N
VF_DBDA	See *Extent Map Faults* on page 3-103.	2121	X		X		X	N
VF_INDA	See *Extent Map Faults* on page 3-103.	2122	X		X		X	N
VF_IXDA	See *Extent Map Faults* on page 3-103.	2123	X		X		X	N
VF_DFDIR	A directory contains the same filename more than once. `verify_vol()` provides the FN and pathname of both files, although in this case the pathnames are identical[6].	2124	X	X	X	X		Y

TABLE 3-3 Fault Summary (Continued)

Mnemonic	Description	Hex	FN1	FN2	PATH 1	PATH 2	BN	Fix-able
VF_IFDIR	A directory entry contains an illegal filename. `verify_vol()` provides the FN and pathname of the file. Note that since the filename is illegal, the last filename in the pathname may not be ASCII.[7]	2125	X			X		Y
VF_FNDIR	A directory entry contains an illegal FN: one that exceeds the allowed maximum. In this case, `fdb_path1` contains the file's pathname while `fdb_fn1` contains the illegal FN. `fdb_fn2` and `fdb_path2` describe the directory containing the illegal entry.[8]	2126	X	X	X	X		Y
VF_INDIR	Incomplete directory entry. The last entry in a directory is a long file name and the end of the directory entry is missing. The FN2 and PATH2 given are of the directory that contains the incomplete directory entry.[9]	2127		X	X	X		Y
VF_BKMU	A single block is used by more than one file.	2128	X	X	X	X	X	N
VF_BBUSE	A bad block is in use.	2129	X		X		X	N
VF_BKFRE	A block that is in use is also marked as free in the volume bitmap.	212A	X		X		X	Y
VF_BBFRE	A bad block is marked as free in the volume bitmap.	212B					X	Y
VF_BKUSE	An unused block is marked as in use in the volume bitmap.	212C					X	Y
VF_INSUFF	Work area too small.	2200						Y

TABLE 3-3 Fault Summary (Continued)

Mnemonic	Description	Hex	FN1	FN2	PATH 1	PATH 2	BN	Fix-able
VF_MAXDEPTH	Directory depth exceeds maximum.	2201						Y
VF_ABORT	Verify routine aborted by user.	2202						Y

1. The file descriptor list, a management block described in the pHILE+ chapter of *pSOSystem System Concepts*.
2. The root block, one of the management blocks described in the pHILE+ chapter of *pSOSystem System Concepts*.
3. File descriptor.
4. File number.
5. verify_vol()If `faultp()` so requests, `verify_vol()` corrects this fault by deleting the second directory entry but not the file to which it refers. That file is still referred to by the first directory entry.
6. If `faultp()` so requests, `verify_vol()` corrects this fault by changing the second filename to `VFN_xxxxxxxx`, where `xxxxxxxx` is the hexadecimal representation of the file's FN. For example, if the file has a FN of 29 (decimal), the filename is set to `VFN_0000001D`.
7. If `faultp()` so requests, `verify_vol()` corrects this fault by changing the filename to `VFN_xxxxxxxx`, where xxxxxxxx is the hexadecimal representation of the file's FN. For example, if the file has an FN of 29 (decimal), the filename is set to `VFN_0000001D`.
8. If `faultp()` so requests, `verify_vol()` corrects this fault by setting the FN to zero (that frees the entry for reuse).
9. If `faultp()` so requests, `verify_vol()` corrects this fault by deleting the incomplete directory entry.

Extent Map Faults

The faults listed in Table 3-4 involve errors in the extent map of a particular file. Recall that the extent map consists of:

- Up to 10 extent descriptors within the file's FD.
- Within the file's FD, an indirect block descriptor that describes an indirect block containing additional extent descriptors.
- Within the file's FD, an index block descriptor that describes an index block containing additional indirect block descriptors.

`verify_vol()` checks for illegal blocks both within an extent and within an indirect or index block descriptor. A block is illegal if its block number is equal to or greater than the number of blocks on the volume. For example, on a volume containing 1000 blocks, any block number greater than 999 is illegal.

A file can be viewed as a sequence of logical blocks numbered from 0. For example, on a volume with 1K blocks, a 4.3 Kbyte file would consist of logical blocks 0 through 4 (logical block 4 being only partly filled.)

Every FD, every indirect block descriptor, and every index block descriptor contains a *last logical block* (LLB) field that indicates the largest logical block number addressed by the associated structure. For example, an indirect block descriptor may have LLB = 200, meaning that the last block in the last extent in the indirect block is the 200th block in the file. `verify_vol()` checks the LLB of every FD, indirect block descriptor, and index block descriptor and reports any inconsistencies.

TABLE 3-4 Extent Map Faults

VF_EXTFD	An FD contains an extent containing an illegal block.
VF_INFD	An FD contains an illegal indirect block number.
VF_IXFD	An FD contains an illegal index block number.
VF_TBCFD	The block count within the FD conflicts with the actual number of blocks in the file.
VF_LLBFD	The LLB in the FD (for the first 10 extents) is incorrect.
VF_LLBIN	The LLB within an indirect block descriptor within an FD is incorrect.
VF_EXTIN	An indirect block contains an extent containing an illegal block.

TABLE 3-4 Extent Map Faults

VF_INIX	An index block contains an illegal indirect block number.
VF_LLBIX	Within an index block, the LLB associated with an indirect block is incorrect.
VF_DBDA	A directory block resides in the data area of the volume.
VF_INDA	An indirect block resides in the data area of the volume.
VF_IXDA	An index block resides in the data area of the volume.

Bad Blocks

A bad block is a block that cannot be read and/or written and therefore cannot be used by pHILE+ file system manager. There are a number of possible strategies for handling bad blocks. One strategy is to mask or redirect them at the driver level so that pHILE+ file system manager never sees them. Another method, which is described here in detail, involves "mapping out" those blocks in the volume's bitmap, so that they are never allocated by pHILE+ file system manager. `verify_vol()` facilitates such modifications to the bitmap.

Recall that each volume contains a bitmap describing which blocks on the volume are in use, and which are free. If the corresponding bit in the map is set to 1, the block is considered to be in use; otherwise, it is considered to be available for allocation by the pHILE+ file system manager when needed. If the bit corresponding to a bad block can be set to 1 before the block is allocated, then the pHILE+ file system manager will never allocate the block, and hence will never read or write it.

To facilitate bad block handling, `verify_vol()` accepts as an input parameter a list of bad blocks. When examining the volume's bitmap, `verify_vol()` expects bits corresponding to these bad blocks to be set to 1, while at the same time expecting the block to be unused. If a bad block is in use, or its corresponding bit is not set, a fault is generated.

The remainder of this section gives a brief outline of a recommended method for handling bad blocks.

There are two types of bad blocks:

Dead Blocks — These blocks are known to be bad prior to volume initialization. They are normally the result of manufacturing defects. Typically, the device manufacturer provides a list of such blocks with each device. They can also be detected by testing the device prior to its initialization.

Failed Blocks — These are blocks that fail some time after the volume has been initialized. They are normally detected in the course of reading or writing the affected block.

Dead blocks are much simpler to handle than failed blocks, because they are detected *before* the pHILE+ file system manager has allocated them. To handle these blocks, perform the following steps:

1. Initialize the volume, taking care not to place the bitmap or FLIST onto any dead blocks (Note: blocks 2 and 3 must not be dead.)
2. Mount the volume and call `verify_vol()`, providing a bad block list containing all dead blocks.
3. Have `faultp()` always return status = 1 (correct fault and continue) for `fdb_code == VF_BBFRE` so that `verify_vol()` will mark the blocks as in use (note that the `"bad block is marked free"` error is correctable).

Failed blocks are harder to handle because the block was already allocated by pHILE+ file system manager before it failed. The block may or may not contain valid data, depending on exactly when the failure occurred. However, since the block is allocated, its corresponding bit is already set. To eliminate such bad blocks, perform the following steps:

1. Add the failed block to the existing list of bad blocks.
2. Invoke `verify_vol()`, which will report `VF_BBUSE`, `bad block is in use` by a particular file.
3. By whatever means, salvage as much of the file as possible, and then delete the file. This returns the bad block to the free block pool and clears its corresponding bit.
4. Invoke `verify_vol()` again. `verify_vol()` now reports `VF_BBFRE`, `"bad block is marked free"`. Now have `faultp()` use return status = 1, to mark the bad block as unavailable for allocation.

Step 3 may be complicated. For example, if a bad block occurs within a directory page, then the entire directory must be deleted after saving as much of its contents as possible.

Note that if `verify_vol()` is to be used to maintain the bitmap in this manner, then an updated list of all bad blocks on the volume must be kept. Integrated Systems suggests that you store the bad block list itself as a file on the volume.

See Also

mount_vol

write_f Writes to an open file.

```
#include <phile.h>
unsigned long write_f(
   unsigned long fid,      /* file identifier */
   void *buffer,           /* output buffer */
   unsigned long bcount    /* output byte count */
   )
```


Volume Types

All volume types except CD-ROM.

Description

`write_f()` writes data into a file. It begins at the current position of the connection's `L_ptr`.

After `write_f()`, the file's `L_ptr` is updated to point to the byte after the last byte written.

This call overwrites the original content of the file. If necessary, `write_f()` expands the file by allocating space to hold the written data.

Arguments

`fid`	Specifies the file identifier associated with the file.
`buffer`	Points to the data to write.
`bcount`	Specifies the number of bytes to write.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. On pHILE+ and MS-DOS volumes, `write_f()` operations are more efficient if `bcount` is an integral multiple of the block size and the L_ptr is positioned at a block boundary.
2. On pHILE+ and MS-DOS volumes, if the requested data includes either entire blocks or a contiguous sequence of blocks and if such blocks are not already in the buffer cache, the pHILE+ file system manager writes these blocks directly from the user's buffer (without going through the buffer cache).
3. `write_f()` automatically positions the `L_ptr` for sequential write operations. If random writes are needed, the `lseek_f()` call should be used to reposition the `L_ptr`.
4. Writing to system or directory files is not allowed.
5. `write_f()` expands a file if space is needed to accommodate the new data.
6. CD-ROM volumes are read-only.

See Also

lseek_f, *sync_vol*, *write_vol*

write_vol

Writes directly to a pHILE+ formatted volume.

```
#include <phile.h>
unsigned long write_vol(
   char *device,          /* volume name */
   unsigned long block,   /* base block */
   unsigned long index,   /* byte offset */
   unsigned long bcount,  /* number of bytes to write */
   void *buffer           /* output buffer */
   )
```


Volume Types

pHILE+ and MS-DOS formatted volumes.

Description

`write_vol()` writes data directly to a pHILE+ formatted volume (bypassing the file system organization imposed by the pHILE+ file system manager).

Arguments

`device`	Points to the null-terminated name of the volume to read.
`block`	Specifies the logical block number where writing begins.
`index`	Specifies where to begin writing within the specified block.
`bcount`	Specifies the number of bytes to write.
`buffer`	Points to the memory area containing the data to write.

Return Value

This system call returns 0 on success or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. If `index` is larger than the volume's block size, the write begins in a subsequent block. For example, on a volume with a 1024-byte block size, writing block 5, index 1224, is the same as writing block 6, index 200.
2. `write_vol()` does not check for the end of the volume; blocks beyond the specified volume size can be written if they physically exist.
3. If the requested data includes either entire blocks or a contiguous sequence of blocks and if such blocks are not already in the buffer cache, the pHILE+ file system manager writes the blocks directly to the volume (without going through the buffer cache.) Therefore, `write_vol()` operations are more efficient when `bcount` and `index` equal integral multiples of blocks.
4. `write_vol()` execution on any block is allowed, including blocks in system files. Therefore, use this call cautiously.
5. CD-ROM volumes are read-only.

See Also

`read_vol`

4 pREPC+ System Calls

This chapter provides detailed information on each system call in the pREPC+ component of pSOSystem. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, a detailed description, its arguments, and its return value. Where applicable, the section also includes the headings "Notes" and "See Also." "Notes" provides important information not specifically related to the call's description, and "See Also" indicates other calls that have related information.

If you need to look up a system call by its functionality, refer to *pREPC+ System Calls*, which lists the calls alphabetically by component and provides a brief description of each call.

pREPC+ error codes are listed in Appendix A, *Error Codes*, For practical reasons, they are not listed here, because every pREPC+ system call can return most or all of the pREPC+ error codes. In addition, errors in other pSOSystem components or device drivers can be reported by pREPC+ system calls.

TABLE 4-1 pREPC+ System Calls

Name	Description	Page
abort	Aborts a task.	4-9
abs	Computes the absolute value of an integer.	4-10
acos	Computes the principal value of the arc cosine of x.	4-11
asctime	Converts the broken-down time to a string.	4-12
asctime_r	(Reentrant) Converts the broken-down time to a string.	4-14
asin	Computes the principal value of the arc sine of x.	4-16
assert	Verifies that a program is operating correctly.	4-17
atan	Computes the principal value of the arc tangent of x.	4-19
atan2	Computes the principal value of the arc tangent of y/x.	4-20
atexit	Registers functions.	4-22
atof	Converts a string to a double.	4-23
atoi	Converts a string to an integer.	4-25
atol	Converts a string to a long integer.	4-27
bsearch	Searches an array.	4-29
calloc	Allocates memory.	4-31
ceil	computes the smallest integral value not less than x.	4-33
clearerr	Clears a stream's error indicators.	4-34
cos	Computes the cosine of x (measured in radians)	4-35
cosh	Computes the hyperbolic cosine of x.	4-36
ctime	Converts the calendar time to a string.	4-37
ctime_r	(Reentrant) Converts the calendar time to a string.	4-39
difftime	Computes the difference between two calendar times.	4-41
div	Performs a division operation on two specified integers.	4-42
errno	The error number returned by the last failing system call.	4-44

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
exit	Terminates a task.	4-45
exp	computes the exponential function of x.	4-47
fabs	Computes the absolute value of a floating-point number x.	4-48
fclose	Closes a stream.	4-49
feof	Tests a stream's end-of-file indicator.	4-51
ferror	Tests a stream's error indicator.	4-52
fflush	Flushes the buffer associated with an open stream.	4-53
fgetc	Gets a character from a stream.	4-55
fgetpos	Gets the current file position indicator for fsetpos.	4-56
fgets	Gets a string from a stream.	4-57
floor	Computes the largest integral value not greater than x.	4-59
fmod	Computes the floating-point remainder of x/y.	4-60
fopen	Opens a file.	4-62
fprintf	Prints formatted output to a stream.	4-66
fputc	Writes a character to a stream.	4-71
fputs	Writes a string to a stream.	4-73
fread	Reads from a stream.	4-75
free	Deallocates memory.	4-77
freopen	Reopens a file.	4-79
frexp	Breaks a floating-point number into a normalized fraction and an integral power of 2.	4-81
fscanf	Reads formatted input from a stream.	4-83
fseek	Sets the file position indicator.	4-87
fsetpos	Sets file position by using the fgetpos result.	4-89

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
ftell	Gets the file position indicator.	4-91
fwrite	Writes to a stream.	4-93
getc	Gets a character from a stream.	4-95
getchar	Gets a character from stdin.	4-96
gets	Gets a string from stdin.	4-97
gmtime	Converts the calendar time to broken-down time.	4-99
gmtime_r	(Reentrant) Converts the calendar time to broken-down time.	4-101
isalnum	Tests for an alphanumeric character.	4-103
isalpha	Tests for an alphabetic character.	4-105
iscntrl	Tests for a control character.	4-107
isdigit	Tests for a digit.	4-109
isgraph	Tests for a graphical character.	4-111
islower	Tests for a lowercase letter.	4-113
isprint	Tests for a printable character.	4-115
ispunct	Tests for a punctuation character.	4-117
isspace	Tests for a space.	4-119
isupper	Tests for an uppercase letter.	4-121
isxdigit	Tests for a hexadecimal digit.	4-123
labs	Computes the absolute value of a long integer.	4-125
ldexp	Multiplies a floating-point number by an integral power of 2.	4-126
ldiv	Performs a division operation on two specified long integers.	4-128
localeconv	Obtains the current locale settings.	4-130
localtime	Converts the calendar time to broken-down time.	4-133
localtime_r	(Reentrant) Converts the calendar time to broken-down time.	4-135

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
log	Computes the natural logarithm of x.	4-137
log10	Computes the base-ten logarithm of x.	4-138
longjmp	Restores the environment set by the most recent invocation of setjmp.	4-139
malloc	Allocates memory.	4-141
mblen	Determines the number of bytes in a multibyte character.	4-142
mbstowcs	Converts a multibyte character string into a wide character string.	4-144
mbtowc	Converts a multibyte character into its wide character equivalent.	4-146
memccpy	Copies characters from one memory area to another.	4-148
memchr	Searches memory for a character.	4-150
memcmp	Compares two objects in memory.	4-152
memcpy	Copies characters in memory.	4-154
memmove	Copies characters in memory.	4-156
memset	Initializes a memory area with a given value.	4-158
mktime	Converts the broken-down time into calendar time.	4-159
modf	Breaks the argument value into integral and fractional parts.	4-162
perror	Prints a diagnostic message.	4-164
pow	Computes x raised to the power y.	4-165
printf	Prints formatted output to stdout.	4-167
putc	Writes a character to a stream.	4-169
putchar	Writes a character to stdout.	4-171
puts	Writes a string to a stream.	4-172
qsort	Sorts an array in ascending order.	4-173

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
rand	Returns a pseudo-random number.	4-175
rand_r	Returns a pseudo-random sequence of integers with repeated calls.	4-176
realloc	Allocates memory.	4-178
remove	Removes a file.	4-180
rename	Renames a file.	4-181
rewind	Resets the file position indicator.	4-183
scanf	Reads formatted input from stdin.	4-185
setbuf	Changes a stream's buffer.	4-187
setjmp	Saves the calling environment for later restoration.	4-189
setlocale	Obtains or changes the program's locale.	4-191
setvbuf	Changes a stream's buffering characteristics.	4-194
sin	Computes the sin of x (measured in radians).	4-196
sinh	Computes the hyperbolic sine of x.	4-197
sprintf	Writes formatted output to a buffer.	4-198
sqrt	Computes the nonnegative square root of x.	4-200
srand	Sets the seed for the random number generator (rand.).	4-201
sscanf	Reads formatted input from a string.	4-203
strcat	Appends one string to another string.	4-205
strchr	Searches a string for a character.	4-206
strcmp	Compares two character strings.	4-207
strcoll	Compares two character strings.	4-208
strcpy	Copies one string to another string.	4-210
strcspn	Calculates the length of a substring.	4-211

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
strerror	Maps an error number to an error message string.	4-212
strftime	Places formatted time and date information into a string.	4-214
strlen	Computes string length.	4-217
strncat	Appends characters to a string.	4-218
strncmp	Compares characters in two strings.	4-220
strncpy	Copies characters from one string to another.	4-222
strpbrk	Searches a string for a character in a second string.	4-224
strrchr	Searches a string for a character.	4-225
strspn	Calculates specified string length.	4-226
strstr	Searches a string for specified characters in another string.	4-227
strtod	Converts a string to a double.	4-228
strtok	Searches a string for tokens.	4-230
strtok_r	Searches a string for tokens.	4-232
strtol	Converts a string to a long integer.	4-234
strtoul	Converts a string to an unsigned long.	4-236
strxfrm	Transforms a string so that it can be used by strcmp().	4-238
tan	Computes the tangent of x (measured in radians).	4-240
tanh	Computes the hyperbolic tangent of x.	4-241
time	Obtains the current calendar time.	4-242
tmpfile	Creates a temporary file.	4-244
tmpnam	Generates a temporary file name.	4-245
tolower	Converts a character to lowercase.	4-247
toupper	Converts a character to uppercase.	4-249
ungetc	Ungets a character.	4-251

TABLE 4-1 pREPC+ System Calls (Continued)

Name	Description	Page
`vfprintf`	Writes formatted output to a stream.	4-253
`vprintf`	Writes formatted output to `stdout`.	4-255
`vsprintf`	Writes formatted output to a buffer.	4-257
`wcstombs`	Converts a wide character string into a multibyte character string.	4-259
`wctomb`	Converts a wide character into its multibyte character equivalent.	4-261

abort

Aborts a task.

```
#include <stdlib.h>
void abort (void)
```

Description

The `abort()` macro is used to terminate a task. `abort()` simply invokes the `exit()` macro with an argument of zero. For further details, refer to the `exit()` macro on page 4-45.

Return Value

If the task is successfully deleted, the `abort()` macro does not return to its caller. If the task cannot be deleted successfully, `abort()` suspends the task indefinitely and does not return to its caller unless the task is explicitly resumed by another task in the system.

Error Codes

None.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

exit

abs

Computes the absolute value of an integer.

```
#include <stdlib.h>
int abs (
   int j   /* long integer */
   )
```

Description

The `abs()` function converts the integer `j` into its absolute value. If the result cannot be represented, the behavior is undefined.

Arguments

`j`	Specifies the integer to be converted.

Return Value

`abs()` returns the absolute value.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`labs`

acos

Computes the principal value of the arc cosine of x.

```
#include <math.h>
double acos (
   double x
   )
```

Description

The acos function computes the principal value of the arc cosine of x. For arguments not in the range [-1, +1], a domain error occurs.

Arguments

x	Specifies a number in the range -1.0 to +1.0, both inclusive.

Return Value

acos returns the arc cosine in the range [0, π] radians.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

cos

asctime

Converts the broken-down time to a string.

```
#include <time.h>
char *asctime (
   const struct tm *timeptr   /* broken-down time */
   )
```

Description

The function `asctime()` converts the broken-down time pointed to by `timeptr` to an equivalent string representation of the form:

```
Sun Jan 1 12:30:13 1995\n\0
```

Arguments

`timeptr`	Points to a `tm` structure that stores the broken-down time. The `tm` structure is defined in the `mktime()` description on page 4-159.

Return Value

The `asctime()` function returns a pointer to the calendar time string.

Error Codes

Refer to Appendix A.

Notes

This function is non-reentrant as it returns a pointer to a statically allocated data area. It is provided only to meet the ANSI requirement. The reentrant version of this function is `asctime_r()`.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

asctime_r, *ctime*, *mktime*, *time*

asctime_r (Reentrant) Converts the broken-down time to a string.

```
#include <time.h>
char *asctime_r (
   const struct tm *timeptr,/* pointer to broken-down time */
   char *buf,               /* result buffer */
   int buflen               /* result buffer length */
   )
```

Description

`asctime_r()` is the reentrant version of the ANSI function `asctime()`, as defined by POSIX 1003.1c. It converts the broken-down time pointed to by `timeptr` to an equivalent string representation of the form:

```
Sun Jan 1 12:30:13 1995\n\0
```

and stores the string in the buffer pointed to by `buf`, which is assumed to have space for at most `buflen` characters. An error may be returned if the converted string contains more than `buflen` characters.

Arguments

`timeptr`	Points to a structure of type `tm` that stores the broken-down time. The `tm` structure is defined in the `mktime()` description on page 4-159.
`buf`	Points to the buffer where `asctime_r()` stores the result.
`buflen`	Specifies the size of `buf`.

Return Value

Upon success, `asctime_r()` returns the value of `buf`. On failure, it returns NULL and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

asctime_r, *ctime*, *ctime_r*, *mktime*, *time*

asin

Computes the principal value of the arc sine of x.

```
#include <math.h>
double asin (
   double x
   )
```

Description

The `asin` function computes the principal value of the arc sine of x. For arguments not in the range [-1, +1], a domain error occurs.

Arguments

x	Specifies a number in the range -1.0 to +1.0, both inclusive.

Return Value

`asin` returns the arc sine in the range [$-\pi/2$, $+\pi/2$] radians.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

sin

assert

Verifies that a program is operating correctly.

```
#include <assert.h>
void assert (
   int expression    /* test expression */
   )
```

Description

The `assert()` macro, defined in the header file `assert.h`, writes error information to `stderr` if the expression `expression` evaluates to zero. The error information includes the text of the argument, the name of the source file, and the source line number. The last two of these are respectively the values of the preprocessing macros `__FILE__` and `__LINE__`.

If `expression` does not evaluate to zero, `assert()` does nothing.

Arguments

`expression`	Specifies the expression to be evaluated.

Return Value

The `assert()` macro returns no value.

Error Codes

None.

Notes

The assert macro is not fully ANSI compliant as it does not invoke `abort()`.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

None.

atan

Computes the principal value of the arc tangent of x.

```
#include <math.h>
double atan (
   double x
   )
```

Description

The atan function computes the principal value of the arc tangent of x.

Arguments

x	Specifies the number whose arc tangent is to be computed.

Return Value

atan returns the arc tangent in the range [-π/2, +p/2] radians.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

tan, *atan2*

atan2

Computes the principal value of the arc tangent of y/x.

```
#include <math.h>
double atan2 (
   double y,
   double x
   )
```

Description

The `atan2` function computes the principal value of the arc tangent of y/x. atan2 uses the signs of both arguments to determine the quadrant of the return value. A domain error can occur if both arguments are zero.

Arguments

x	The denominator of the number whose arc tangent is to be computed.
y	The numerator of the number whose arc tangent is to be computed.

Return Value

`atan2` returns the arc tangent in the range [-π, +π] radians.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

tan, *atan*

atexit

Registers the function(s) pointed to by `func`.

```
#include <stdlib.h>
int atexit(
   void (*func) (void)
   )
```

Description

The `atexit` function registers the function pointed to by `func`, to be called without arguments when the `exit()` function is called.

Arguments

`func` The function(s) to be registered.

Return Value

This function returns zero if the registration succeeds, and nonzero if it fails.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

exit

atof

Converts a string to a double.

```
#include <stdlib.h>
double atof(
   const char *nptr   /* string */
   )
```

Description

This function converts the initial part of the string pointed to by `nptr` to a `double` representation. Leading white spaces are ignored. The argument `nptr` can be in scientific exponential form (for example, +123.45e+67, -123.45E+67). This function stops parsing `nptr` when it detects a character inconsistent with a `double` data type. If the first nonwhite space character is other than a sign, a digit or a decimal point, a value of 0 is returned.

Except for the behavior on error, this call is equivalent to:

```
strtod(str, (char **)NULL);
```

Arguments

`nptr`	Points to the string to be converted.

Return Value

This function returns the converted value. In the event of an error, `errno` is set to indicate the condition.

Error Codes

Refer to Appendix A.

Notes

The pREPC+ library returns `double` values (including floating point) in the CPU register pair designated by the compiler to receive a return value of type `double` from a function call when a hardware floating point is *not* selected. Please refer to your compiler manual for the register pair. Additionally, if the FPU bit is set in the processor type entry of the Node Configuration Table, the pREPC+ library also places the floating point value in the floating point register designated by the com-

piler to receive a return value of type `double` when a hardware floating point *is* selected.

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

strtod

atoi

Converts a string to an integer.

```
#include <stdlib.h>
int atoi(
   const char *nptr   /* string */
   )
```

Description

The `atoi()` function converts the initial part of the string pointed to by `nptr` to an `int` representation. Leading white spaces are ignored. The conversion terminates when a nondigit character is detected. If the first nonwhite space character is not a digit, a value of 0 is returned.

Except for the behavior on error, this call is equivalent to:

```
(int) strtol(str, (char **)NULL, 10);
```

Arguments

`nptr` Points to the string to be converted.

Return Value

This function returns the converted value. If an error occurs, `errno` is set to indicate the condition.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

strtol

atol

Converts a string to a long integer.

```
#include <stdlib.h>
long atol(
   const char *nptr   /* string */
   )
```

Description

The `atol()` function converts the initial part of the string pointed to by `nptr` to a `long int` representation. Leading white spaces are ignored. The conversion terminates when a nondigit character is detected. If the first non-whitespace character is not a digit, a value of 0 is returned.

Except for the behavior on error, this call is equivalent to:

```
strtol(str, (char **)NULL, 10);
```

Arguments

`nptr`	Points to the string to be converted.

Return Value

This function returns the converted value. If an error occurs, `errno` is set to indicate the condition.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

strtol

bsearch Searches an array.

```
#include <stdlib.h>
void *bsearch(
   const void *key,    /* search key */
   const void *base,   /* start point */
   size_t nmemb,       /* number of members */
   size_t size,        /* member size */
   int (*compar)(const void *, const void *)
                       /* comparison operator */
   )
```


Description

The bsearch() function searches an array of nmemb objects, the initial element of which is pointed to by base, for an element that matches the object pointed to by key. The size of each element of the array is specified by size.

The array must be sorted in ascending order. A user supplied comparison function, compar, is called by bsearch with two arguments. The first argument to compar is a pointer to the key object, and the second is a pointer to an array member. The compar function must return an integer that is either less than, equal to, or greater than zero if key object is considered, respectively, to be less than, equal to, or greater than the array member.

Arguments

key	Points to the object to be matched in the search.
base	Points to the beginning of the array to be searched.
nmemb	Specifies the number of members in the array.
size	Specifies the size of each member in the array.

Return Value

This function returns a pointer to the first matching member detected. If no match is found, it returns a null pointer. In the event of an error, errno is set to indicate the condition.

Error Codes

Refer to Appendix A.

Notes

The `compar` function can call a limited set of pREPC+ functions. These functions consist of all character handling functions and all string handling functions except `strtok()`. Other pREPC+ functions cannot be called from `compar`.

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`qsort`

calloc Allocates memory.

```
#include <stdlib.h>
void *calloc(
   size_t nmemb,   /* number of allocation units */
   size_t size     /* size of allocation unit */
   )
```

Description

The `calloc()` function allocates memory for `nmemb` data objects, each of whose size is specified by `size`. The allocated memory is initialized to 0.

Arguments

`nmemb`	Specifies the number of data objects for which `calloc()` allocates memory.
`size`	Specifies the size of each data object.

Return Value

This function returns a pointer to the memory allocated, or a null pointer if no memory is allocated. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

1. `calloc()` calls the pSOS+ region manager to allocate the memory.
2. Memory is always allocated from Region 0.
3. The caller can be blocked if memory is not available and the wait option is selected in the pREPC+ Configuration Table.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

free, *malloc*, *realloc*

ceil

Computes the smallest integral value not less than x.

```
#include <math.h>
double ceil (
   double x
   )
```

Description

The `ceil` function computes the smallest integral value not less than x, expressed as a double.

Arguments

x	The number to which the ceiling function is applied.

Return Value

`ceil` returns the smallest integral value not less than x.

Error Codes

None

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

floor

clearerr Clear's a stream's error indicators.

```
#include <stdarg.h>
#include <stdio.h>
void clearerr(
   FILE *stream   /* stream pointer */
   )
```

Description

The `clearerr()` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function does not return a value. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

COS

Computes the cosine of x (measured in radians).

```
#include <math.h>
double cos (
   double x
   )
```

Description

The COS function computes the principal value of the cosine of x (measured in radians).

Arguments

x	The angle, expressed in radians, whose cosine is to be computed.

Return Value

COS returns the cosine of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

cosh, *acos*

cosh Computes the hyperbolic cosine of x.

```
#include <math.h>
double cosh (
   double x
   )
```

Description

The cosh function computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Arguments

x	The angle whose hyperbolic cosine is to be computed.

Return Value

cosh returns the hyperbolic cosine of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

cos, *acos*

ctime

Converts the calendar time to a string.

```
#include <time.h>
char *ctime (
   const time_t *timer   /* calendar time */
   )
```

Description

The `ctime()` function converts the calendar time pointed to by `timer` to a string representation of the form:

Sun Jan 1 12:30:13 1995\n\0

The time is represented in local time. This call is equivalent to:

```
asctime(localtime(timer))
```

The calendar time is generally obtained through a call to `time()`.

The buffer used by `ctime()` to hold the formatted output string is a statically allocated character array and is overwritten each time the function is called. To save the contents of the string, you need to copy it elsewhere.

Arguments

`timer`	Points to the calendar time.

Return Value

The `ctime()` function returns the pointer to the converted calendar time string.

Error Codes

Refer to Appendix A.

Notes

This function is non-reentrant as it returns a pointer to a statically allocated data area. It is provided only to meet the ANSI requirement. The reentrant version of this function is `ctime_r()`.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

asctime, *asctime_r*, *ctime_r*, *mktime*, *time*

ctime_r (Reentrant) Converts the calendar time to a string.

```
#include <time.h>
char *ctime_r (
   const time_t *timer,    /* calendar time */
   char *buf,              /* result buffer */
   int buflen              /* result buffer length */
   )
```


Description

`ctime_r()` is the reentrant version of the ANSI function `ctime()`, as defined by POSIX 1003.1c. It converts the calendar time pointed to by `timer` to a string representation of the form:

```
Sun Jan 1 12:30:13 1995\n\0
```

The time is represented in local time. `ctime_r()` stores the string in the buffer pointed to by `buf`, which is assumed to have space for at most `buflen` characters. An error may be returned if the converted string contains more than `buflen` characters.

The calendar time is generally obtained through a call to `time()`.

Arguments

`timer`	Points to the calendar time.
`buf`	Points to the buffer where `ctime_r()` stores the result.
`buflen`	Specifies the size of `buf`.

Return Value

Upon success, `asctime_r()` returns the value of `buf`. On failure, it returns NULL and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

asctime, *asctime_r*, *ctime*, *mktime*, *time*

difftime

Computes the difference between two calendar times.

```
#include <time.h>
double difftime (
   time_t time1,   /* finish time */
   time_t time0    /* start time */
   )
```

Description

The `difftime()` function computes the difference, in seconds, between two calendar times: `time1` - `time0`.

Arguments

`time1` Specifies the finish time.

`time0` Specifies the start time.

Return Value

The `difftime()` function returns the difference expressed in seconds as a `double`.

Error Codes

None.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

time

div

Performs a division operation on two specified integers.

```
#include <stdlib.h>
div_t div (
   int numer,    /* numerator */
   int denom     /* denominator */
   )
```

Description

The `div()` function computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot * denom + rem` is equal to `numer`.

Arguments

`numer`	Specifies the numerator.
`denom`	Specifies the denominator.

Return Value

The `div()` function returns a structure of type `div_t`. This structure is defined in `stdlib.h` as follows:

```
typedef struct {
   int quot;   /* the quotient */
   int rem;    /* the remainder */
   } div_t;
```

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

ldiv

errno

The error number returned by the last failing system call.

```
#include <errno.h>
int errno
```

Description

The `errno` is a macro which expands to a modifiable `lvalue` that has type `int`. Its value can be set to a positive number by several library or system calls. The macro is defined in the header file `errno.h`. It returns the error number from the last failing system call or library function.

If the macro definition is suppressed (by using `#undef` pre-processor directive), or an application defines an identifier with the name `errno`, the behavior is undefined.

The value of `errno` is zero at task startup, but is never set to zero by any library function or system service.

Return Value

This macro returns the current value of `errno` for the calling task.

Error Codes

Refer to Appendix A.

Notes

`errno` generates a call to the pSOS+ `errno_addr()` system service.

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = NO`

exit

Terminates a task.

```
#include <stdlib.h>
void exit(
   int status   /* termination status */
   )
```

Description

The `exit()` macro terminates the task that invokes it. `exit()` prints an error message on the task's standard error stream if a non-zero `status` is passed to it. It then executes the task exit sequence described in `t_delete()` on page 2-208, with the exception that it does not automatically invoke "close" functions for pSOSystem components other than pREPC+ (see *Notes*). If the task cannot be deleted successfully, `exit()` suspends the task indefinitely.

Arguments

`status`	Contains the termination status printed by the error message.

Return Value

If the task is successfully deleted, the `exit()` macro does not return to its caller. If the task cannot be deleted successfully, `exit()` suspends the task indefinitely and does not return to its caller unless the task is explicitly resumed by another task in the system.

Error Codes

None.

Notes

If you have pSOSystem components other than pREPC+ configured in your system, you need to edit the definition of `exit()` in `stdlib.h` to uncomment the necessary "close" functions in the task exit sequence. The "close" functions release any task-specific resources held by pSOSystem components.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

abort

exp

Computes the exponential function of x.

```
#include <math.h>
double exp (
   double x
   )
```

Description

The exp function computes the exponential function of x.

Arguments

x	The number to which the exp function is applied.

Return Value

exp returns the exponential value of x.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

log, *log10*

fabs

Computes the absolute value of a floating-point number x.

```
#include <math.h>
double fabs (
   double x
   )
```

Description

The `fabs` function computes the absolute value of a floating-point number x.

Arguments

x	Number whose absolute value is to be computed.

Return Value

`fabs` returns the absolute value of `x`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

abs

fclose Closes a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fclose(
   FILE *stream   /* stream pointer */
   )
```

Description

The `fclose()` function first flushes the buffer associated with the stream pointed to by `stream` and closes the associated file or I/O device. Any unwritten buffered data is written to the associated file or I/O device, and any unread buffered data is discarded. The stream is disassociated from the file or I/O device.

If the buffer was automatically allocated when the stream was opened, it is reclaimed by the system. The user is responsible for returning user-supplied buffers.

When invoked with a null stream pointer, `fclose()` has a special significance under pREPC+. It causes pREPC+ to close all streams opened by the calling task.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function returns 0 if successful or end-of-file (`EOF`) if an error occurs. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

1. pREPC+ calls the pSOS+ function `de_close()` if the stream was associated with an I/O device.
2. pREPC+ calls the pHILE+ function `close_f()` if the stream was associated with a disk file.

3. If a task uses any of the pREPC+ functions, it must call `fclose(0)` to release all pREPC+ resources prior to deleting itself through a `t_delete()` system call. Refer to the `t_delete()` call description on page 2-208 for the complete exit sequence.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fopen

feof

Tests a stream's end-of-file indicator.

```
#include <stdarg.h>
#include <stdio.h>
int feof(
   FILE *stream   /* stream pointer */
   )
```

Description

The `feof()` function tests the end-of-file indicator for the stream pointed to by `stream`.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function returns a nonzero number if the end-of-file indicator is set for `stream` and zero if it is not set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

ferror Tests a stream's error indicator.

```
#include <stdarg.h>
#include <stdio.h>
int ferror(
   FILE *stream   /* stream pointer */
   )
```

Description

This function tests the error indicator for the stream pointed to by `stream`.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function returns a nonzero number if the error flag is set and zero if it is not set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

fflush

Flushes the buffer associated with an open stream.

```
#include <stdarg.h>
#include <stdio.h>
int fflush(
   FILE *stream   /* stream pointer */
   )
```

Description

The `fflush()` function writes any unwritten, buffered data associated with the stream pointed to by `stream` to the file or I/O device. An error is returned if the stream has not been opened for write or update. If `stream` is a null pointer, the `fflush()` function performs the flushing action on all the streams open for write or update.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function returns 0 if successful or `EOF` on error. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

1. If the write is to an I/O device, `fflush()` calls the pSOS+ I/O call `de_write()`.
2. If the write is to a disk file, `fflush()` calls the pHILE+ call `write_f()`.

Callable From

- Task

Required SC_PREPC Setting

```
SC_PREPC = YES
```

fgetc

Gets a character from a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fgetc(
   FILE *stream   /* stream pointer */
   )
```

Description

The `fgetc()` function reads the next character, as an `unsigned char` converted to an `int`, from the input stream pointed to by `stream` and advances the associated file position indicator for the stream, if defined. It is operationally equivalent to the `getc` function.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

This function returns the character that is read from the stream. If the end-of-file condition is detected, the stream's end-of-file indicator is set and `EOF` is returned. If a read error occurs, the stream's error indicator is set, `EOF` is returned, and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

fgetpos

Gets the current file position indicator for `fsetpos`.

```
#include <stdarg.h>
#include <stdio.h>
int fgetpos(
   FILE *stream,   /* stream pointer */
   fpos_t *pos     /* stream position */
   )
```

Description

The `fgetpos()` function stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `pos`. This value can be used by the `fsetpos()` function to reposition the file position indicator of the stream to its position at the time of the call to the `fgetpos()` function.

Arguments

`stream`	Points to an open pREPC+ stream.
`pos`	Points to the object where `fgetpos()` stores the current file position indicator.

Return Value

If successful, this function returns a zero. If not successful or if `stream` references an I/O device, this function returns an `EOF` and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = YES`

fgets

Gets a string from a stream.

```
#include <stdarg.h>
#include <stdio.h>
char *fgets(
   char *s,        /* buffer */
   int n,          /* length */
   FILE *stream    /* stream pointer */
   )
```


Description

The `fgets()` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream`. The characters go into the user buffer pointed to by `s`. The function stops reading characters when a new-line character (which is retained) or an end-of-file condition is detected. A null character is written immediately after the last character is read into the user buffer.

If `stream` references a disk file, its position indicator is advanced.

Arguments

`s`	Points to the user buffer where `fgets()` stores characters.
`n`	Specifies the number of characters to read, plus one for the null terminator.
`stream`	Points to an open pREPC+ stream.

Return Value

This function returns `s` if successful. If a read error occurs or an end-of-file condition is detected before any characters are read, a null pointer is returned and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fputs

floor

Computes the largest integral value not greater than x.

```
#include <math.h>
double floor (
   double x
   )
```

Description

The `floor` function computes the largest integral value not greater than x.

Arguments

x	The number to which the floor function is applied.

Return Value

`floor` returns the largest integral value not greater than x, expressed as a double.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

ceil

fmod

Computes the floating-point remainder of x/y.

```
#include <math.h>
double fmod (
   double x,
   double y
   )
```

Description

The fmod function computes the floating-point remainder of x/y.

Arguments

x	The numerator of the input number.
y	The denominator of the input number. It should be a non-zero number.

Return Value

fmod returns the value x-i*y, for some integer i. If y is nonzero, the result has the same sign as x and the magnitude less than the magnitude of y. If y is zero, a domain error occurs.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

modf

fopen

Opens a file.

```
#include <stdarg.h>
#include <stdio.h>
FILE *fopen(
   const char *filename,   /* file name */
   const char *mode        /* access mode */
   )
```

Description

The `fopen()` function opens a disk file or I/O device whose name is specified by the string pointed to by `filename` and associates a stream with it.

`fopen()` allocates a `FILE` structure for the opened stream. It contains control information for the opened I/O device or disk file. `fopen()` returns a pointer to the allocated FILE structure that subsequent calls require to perform various I/O operations on the I/O device or disk file (for example, for an `fread` or `fwrite` call).

Disk files have position indicators that determine where the next byte is read from or written to in the file. Position indicators have no meaning for I/O devices.

Arguments

`filename`	Points to the name of the disk file or I/O device to be opened.	
`mode`	Points to a string that specifies the mode in which the file is to be opened. The `mode` string must begin with one of the following sequences:	
	`r`	Open text file for reading.
	`w`	Truncate to zero length or create text file for writing.
	`a`	Append: open or create text file for writing at the end-of-file.
	`rb`	Open binary file for reading.
	`wb`	Truncate to zero length or create binary file for writing.
	`ab`	Append: open or create a binary file for writing at the end-of-file.

`r+`	Open text file for updating, reading and writing at the current file position.
`w+`	Truncate to zero length or create text file for updating, reading and writing at the current file position.
`a+`	Append: open or create text file for updating, reading at the current file position and writing at the end-of-file.
`r+b` or `rb+`	Open binary file for updating, reading and writing at the current file position.
`w+b` or `wb+`	Truncate to zero length or create binary file for updating, reading and writing at the current file position.
`a+b` or `ab+`	Append: open or create a binary file for updating, reading at current file position and writing at the end-of-file.

Opening a disk file with read mode (`r` as the first character in `mode` argument) fails if the file does not exist or cannot be read.

Opening a disk file with append mode (`a` as the first character in `mode` argument) causes all subsequent writes to the then current `EOF`, regardless of intervening calls to the `fseek` function.

When a disk file is opened with update mode (`+` as the second or third character in `mode` argument) both read and write operations may be performed on the associated stream. However, output may not be directly followed by input, or vice-versa, without an intervening call to the `fflush` function or to a file positioning function, via `fseek`, `fsetpos` and `rewind`. The only exception to the above rule is a read operation that encounters end-of-file.

Return Value

If the open operation is successful, this function returns a pointer to the `FILE` object that must be used in subsequent calls to specify this opened stream. If the operation fails, a null pointer is returned and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

1. The disk files are managed by the pHILE+ file system manager and are designated by pHILE+ pathnames (for example, `0.1/abc`). The I/O devices are identified by pSOS+ logical device numbers represented as a string of the form `M.N`, where `M` is the major number and `N` is the minor number of the I/O device (for example, 0.1). When reading or writing disk files, the pREPC+ library calls the pHILE+ file system manager. When reading or writing I/O devices, the pREPC+ library calls the pSOS+ I/O Supervisor directly.

2. `fopen()` internally makes a call to the pHILE+ `open_f` function when it opens a disk file and the pSOS+ `de_open` function when it opens an I/O device.

3. If the volume is an NFS volume, two conditions can cause problems related to the file mode. The conditions are as follows:

   a. A file that did not previously exist is created in an NFS volume by the `fopen()` call with a UNIX-like privilege mode automatically set to 0x180 (octal 600/rw for the user.) If, for example, the mode in the `fopen()` call is manually specified as `w`, a conflict could result. The pREPC+ library allows only file operations that are valid for the specified mode. Therefore (using the preceding example), the pREPC+ library does not allow a read operation on the following file:

      ```
      fopen("0.0/file1.dat", "w");
      ```

   b. The restrictions placed on file operations depend on the mode extended to files that exist prior to the `fopen()` call. For example:

      ```
      fopen("0.0/file2.dat", "r");
      ```

      succeeds if `file2.dat` exists prior to the `fopen()` call. A subsequent read operation would fail if that file had an access mode (under NFS) that did not allow a user-read. Currently, no method exists under the pREPC+ library to change the access mode of an NFS file.

4. Since the underlying mechanism (pHILE+ and pSOS+ I/O device manager) do not, as of yet, support the concept of text files, pREPC+ treats text files as binary files. However, for forward compatibility, you must specify the “b” character in the mode string if the file being opened through `fopen` contains binary data that should be read or written without performing any translations on it.

5. Though native MS-DOS systems differentiate between text and binary files, the pHILE+ implementation of MS-DOS file system does not.

6. For I/O devices, the text streams are treated as binary streams.

7. For I/O devices, the operations of truncating, creating and appending are meaningless. In modes starting with `w` or `a`, an error is returned if the device being opened is not configured into the system. Also, the append mode is treated the same as the write mode for I/O devices.

8. Opening a device that does not support random access in update mode (e.g., serial I/O), though allowed by pREPC+, does not make sense. None of the devices supported by pREPC+ support random access and hence none of them shall be opened in update mode. To perform read and write to a single device, two separate streams, one in read mode and the other in write mode, shall be opened instead.

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = YES`

See Also

`fclose`, *`fseek`*

fprintf

Prints formatted output to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fprintf(
   FILE *stream,          /* stream pointer */
   const char *format,    /* format control */
   ...                    /* arguments 1 through n */
   )
```

Description

The `fprintf()` function writes output to the stream pointed to by `stream`. A format control string, pointed to by `format`, specifies how the subsequent arguments are converted for output.

Arguments

`stream` Points to an open pREPC+ stream.

`format` Points to the format control string. The `format` string consists of ordinary characters (except the % character) and conversion specifications. The ordinary characters are simply copied to the output stream. The conversion specifications determine the form of the arguments' output. Each argument should have one conversion specification.

Each conversion specification begins with a % character and ends with a conversion specification character. One or more of the following can be positioned between the % and ending specification character, in the order specified below:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described below, has been given) to the field width. The field width takes the form of an asterisk * (described below) or a decimal integer.

- An optional *precision* that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point character for e, E, and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional modifier `h`, `l` (ell), or `L` indicating the size of the receiving object. For instance, the conversion specifier `d` is preceded by `h` if the corresponding argument is a `short int` rather than an `int` (the argument will have been promoted according to the rules of integral promotions, and its value will be converted to `short int` before printing.) A table of modifiers is presented below, under the list of conversion specifiers.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The argument specifying field width, or precision, or both, should appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

`-` The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)

`+` The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

`space` If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the space and `+` flags both appear, the space flag will be ignored.

`#` The result is converted to an "alternate form". For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (or `X`) conversion, a nonzero result will have `0x` (or `0X`) prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

`d` or `i` The argument is assumed to be an `int` and is converted to signed decimal notation. The precision specifies the minimum number of digits that appear.

`o` The argument is assumed to be an unsigned `int` and is converted to unsigned octal notation. The precision specifies the minimum number of digits to appear.

`u` The argument is assumed to be an unsigned `int` and is converted to unsigned decimal notation. The precision specifies the minimum number of digits to appear.

`x` or `X` The argument is assumed to be an unsigned `int` and is converted to hexadecimal notation. The precision specifies the minimum number of digits to appear.

`f` The argument is assumed to be a double and is converted to decimal notation in the form [-]ddd.ddd. The precision specifies the number of digits to appear after the decimal point. The default precision is six. If the precision is zero, no decimal-point is printed. The value is rounded to the appropriate number of digits.

`e` or `E` The argument is assumed to be a double and is converted to decimal notation in the form [-]d.ddde(E)+dd. The precision specifies the number of digits to appear after the decimal-point. The default precision is six. If the precision is zero, no decimal-point is printed. The value is rounded to the appropriate number of digits.

`g or G` The argument is assumed to be a double and is converted to decimal notation in the form of either `e` or `f`. This depends on the value of the converted number. The `f` form is used unless the exponent is less than -4 or greater than or equal to the precision. The precision specifies the number of significant digits. The decimal-point character appears only if a digit follows it.

`c` The argument is assumed to be an `int` and is converted to an `unsigned char`.

`s` The argument is assumed to be a pointer to a string. Characters in the string are printed until a null character is detected or until the number of characters indicated by the precision is exhausted.

`p` The argument is assumed to be a pointer to a void and the value of the pointer is printed as a hexadecimal number.

`n` The argument is assumed to be pointer to an integer into which is written the number of characters written by this call so far.

`%` A % character is written.

Below is a table of modifiers that can precede a conversion specifier. If a modifier appears with any conversion specifier not listed, the behavior is undefined.

Modifier	Specifier	Default Argument Type	Modified Argument Type
h	d,i	int	short int
h	o,u,x,X	unsigned int	unsigned short
h	n	pointer to int	pointer to short int
l	d,i	int	long
l	o,u,x,X	unsigned int	unsigned long
l	n	pointer to int	pointer to long int
L	e,E,f,g,G	double	long double

`...` Arguments 1 through n are written by f`printf()` according to the specifications contained in the format control string.

Return Value

This function returns the number of characters written. It returns a negative number if a write error occurs and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

fputc

Writes a character to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fputc(
   int c,          /* character */
   FILE *stream    /* stream pointer */
   )
```


Description

The `fputc()` function writes the character specified by `c` to the output stream pointed to by `stream` after converting it to an `unsigned char`. This function operates the same as the `putc` function.

If `stream` designates a disk file, its position indicator is advanced appropriately.

Arguments

`c`	Specifies the character to write.
`stream`	Points to an open pREPC+ output stream.

Return Value

This function returns the character written. If a write error occurs, the stream's error indicator is set, EOF is returned and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fgetc

fputs

Writes a string to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fputs(
   const char *s,    /* string */
   FILE *stream      /* stream pointer */
   )
```


Description

The `fputs()` function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null character is not written.

Arguments

`s`	Points to the string to be written.
`stream`	Points to an open pREPC+ stream.

Return Value

This function returns zero if the operation succeeds and EOF if it fails. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

See Also

`fgets`

Notes

Callable From

- Task

Required SC_PREPC Setting

```
SC_PREPC = YES
```

fread

Reads from a stream.

```
#include <stdarg.h>
#include <stdio.h>
size_t fread(
   void *ptr,     /* buffer */
   size_t size,   /* element size */
   size_t nmemb,  /* element count */
   FILE *stream   /* stream pointer */
   )
```


Description

This function reads up to nmemb elements whose size is specified by size from the stream pointed to by stream and puts them into the user buffer pointed to by ptr. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of stream's file position indicator is indeterminate. If a partial item is read, its value is indeterminate.

Arguments

ptr	Points to the user buffer where items are stored.
size	Specifies the size of each item.
nmemb	Specifies the number of items to read.
stream	Points to an open pREPC+ stream.

Return Value

This function returns a count of the number of items successfully read. If this value is less than nmemb, then one of two conditions occurred:

- A read error occurred. The stream's error indicator and errno are set.
- You reached the end of the file.

Check the EOF flag in the file structure using the macro provided in stdio.h. If this macro returns TRUE (1), then you have reached the end of the file. Otherwise, check errno.

If `size` or `nmemb` is zero, `fread()` returns 0 and the contents of the array and the state of the stream remain unchanged.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

`fwrite`, *`fopen`*

free

Deallocates memory.

```
#include <stdlib.h>
void free(
   void *ptr   /* pointer to memory segment */
   )
```

Description

This function deallocates a specified memory segment (`ptr`). If the memory segment was not previously allocated by `calloc()`, `malloc()` or `realloc()`, or if the space has been deallocated by a call to `free()` or `realloc()`, the results are unpredictable.

When invoked with a memory segment pointer of value -1, `free()` has a special significance under pREPC+. It causes pREPC+ to reclaim all memory allocated by pREPC+ on behalf of the calling task, either implicitly or explicitly by calls to the `calloc()`, `malloc()` or `realloc()` functions.

Arguments

`ptr` Points to the memory segment to deallocate.

Return Value

This function does not return a value. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

`free()` calls the pSOS+ region manager to deallocate the memory.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

calloc, *malloc*, *realloc*

freopen Reopens a file.

```
#include <stdarg.h>
#include <stdio.h>
FILE *freopen(
   const char *filename,   /* filename */
   const char *mode,       /* access mode */
   FILE *stream            /* stream pointer */
   )
```


Description

The `freopen()` function first closes the file specified by `stream`. Then it opens the file named by the string pointed to by `filename` and associates it with the stream pointed to by `stream`. The file is opened in the mode specified by `mode`, which is interpreted just as in `fopen`. The error and end-of-file indicators for the stream are cleared.

If the close operation fails, `filename` is still opened and attached to `stream`.

This call can be used to rename `stdin`, `stdout`, and `stderr`.

Arguments

`filename`	Points to the name of the file to be opened.
`mode`	Points to the mode string, which specifies how to open the file.
`stream`	Points to an open pREPC+ stream.

Return Value

This function returns a file pointer if the file specified by `filename` is opened, or it returns a null pointer if it does not open the file. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fopen, *fclose*

frexp

Breaks a floating-point number into a normalized fraction and an integral power of 2.

```
#include <math.h>
double frexp (
   double value,
   int *exp
   )
```


Description

The `frexp` function breaks a floating-point number into a normalized fraction and an integral power of 2. The integer is stored in the int object pointed to by exp.

Arguments

`value`	Floating-point number to be broken up.
`exp`	Pointer to where the integral part of the answer is returned.

Return Value

`frexp returns the integral part of value` in a location pointed to by `exp`. If value is zero, both parts of the result are zero.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

ldexp

fscanf

Reads formatted input from a stream.

```
#include <stdarg.h>
#include <stdio.h>
int fscanf(
   FILE *stream,          /* stream pointer */
   const char *format,    /* format control string */
   ...                    /* arguments 1 through n */
   )
```


Description

The `fscanf()` function reads input from the stream specified by `stream`. As input is read, it is divided into input fields. An input field is defined as a string of non-white space characters. It extends either to the next white space character or up to a specified field width. The input fields are handled in a manner determined by a format control string. The input fields are converted to data items and stored in variables pointed to by the remaining arguments. If insufficient arguments are provided for `format`, the behavior is undefined. If `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Arguments

`stream` Points to an open pREPC+ stream.

`format` Points to the format control string. The format is a multibyte character sequence, beginning and ending with its initial shift state. It is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional modifier `h` or `l` (ell), indicating the size of the receiving object. For instance, the conversion specifier `d` is preceded by `h` if the corresponding argument is a pointer to a `short int` rather than a pointer to an `int`. A table of modifiers is presented below, under the list of conversion specifiers.

- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

`fscanf()` executes each directive of the format in turn. If a directive fails, as detailed below, `fscanf()` returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the `isspace()` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier.

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The conversion specifier characters and their definitions are as follows:

`d, i` An optionally signed decimal integer is expected. The corresponding argument should be a pointer to an integer.

`o` An optionally signed octal integer is expected. The corresponding argument should be a pointer to an integer.

`u` An optionally signed octal integer is expected. The corresponding argument should be a pointer to an unsigned long integer.

`x` An optionally signed hexadecimal integer is expected. The corresponding argument should be a pointer to an integer.

`e,f,g` An optionally signed floating-point number is expected. The corresponding argument should be a pointer to a float.

`p` An unsigned hexadecimal number is expected. The corresponding argument should be a pointer to a pointer to void.

`s` A sequence of non-white-space characters is expected. The corresponding argument should be a pointer to a character array large enough to accept the sequence and an added, terminating null character.

`c` A sequence of characters is expected. The number of characters in the sequence should be equal to the field width. If the field width is not specified, one character is expected. The corresponding argument should be a pointer to a character array large enough to accept the sequence. A terminating null character is not added.

`[` A sequence of characters is expected. Every character must match one of the characters listed after the [character and up to and including a] character. If the first character listed after the initial [character is a circumflex (^) character, then the characters read must not match the characters given in the list. A character list beginning with [] or [^] is a special case. If this occurs, the first] character does not end the list and a second] character is needed. The corresponding argument should be a pointer to a character array large enough to accept the sequence and an added, terminating null character.

`n` No input is read. The corresponding argument should be a pointer to an integer that is loaded with the number of characters read so far.

`%` A % character is expected. No assignment occurs.

Below is a table of the modifiers that can precede a conversion specifier. If a modifier appears with any conversion specifier not listed, the behavior is undefined.

Modifier	Specifier	Default Argument Type	Modified Argument Type
h	d,i,n	pointer to int	pointer to short
h	o,x,u	pointer to unsigned int	pointer to unsigned short
l	d,i,n	pointer to int	long
l	o,x,u	pointer to unsigned int	pointer to unsigned long
l	e,f,g	pointer to float	pointer to double

The L modifier is not supported by fscanf().

... Arguments 1 through n point to variables where input is stored.

Return Value

This function returns EOF and sets errno if an input failure occurs before any conversion. Otherwise, it returns the number of input items assigned, which can be fewer than provided, even zero, in the event of an early matching failure.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

scanf

fseek

Sets the file position indicator.

```
#include <stdarg.h>
#include <stdio.h>
int fseek(
   FILE *stream,   /* stream pointer */
   long offset,    /* file offset */
   int whence      /* relative file base */
   )
```


Description

The `fseek()` function sets the file position indicator for the stream pointed to by `stream`.

For a text stream, either `offset` should be zero, or `offset` should be a value returned by an earlier call to `ftell` function on the same stream and `base` should be `SEEK_SET`.

A successful call to `fseek()` clears the stream's end-of file indicator and undoes any effect of `ungetc` function on the same stream.

If `stream` refers to an I/O device, this function does nothing and returns a zero.

Arguments

`stream` Points to an open pREPC+ stream.

`offset` For a binary stream, specifies the offset value, which will be added to the position specified by `whence` to calculate the new value of the position indicator.

`whence` Specifies the relative file base. `whence` can have one of the following values:

Value	Description
`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current position in file
`SEEK_END`	End of file

Return Value

This function returns zero if the operation is successful or a nonzero number if unsuccessful. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

ftell, *fsetpos*, *fgetpos*

fsetpos

Sets file position by using the fgetpos result.

```
#include <stdarg.h>
#include <stdio.h>
int fsetpos(
   FILE *stream,       /* stream pointer */
   const fpos_t *pos   /* stream position */
   )
```


Description

The fsetpos() function sets the position indicator for the stream pointed to by stream to the value of the object pointed to by pos.

A successful call to fsetpos function clears the EOF indicator for the stream and undoes any effects of the ungetc function on the same stream.

Arguments

stream	Points to an open pREPC+ stream. If stream refers to an I/O device, this function does nothing and returns a 0.
pos	Points to an object that specifies the new value of the position indicator. The object should contain a value previously returned by the fgetpos() function on the same stream.

Return Value

This function returns a 0 if successful and a nonzero number if unsuccessful. If an error occurs, errno is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fgetpos

ftell

Gets the file position indicator.

```
#include <stdarg.h>
#include <stdio.h>
long ftell(
   FILE *stream   /* stream pointer */
   )
```

Description

The `ftell()` function obtains the current value of the position indicator for the stream pointed to by `stream`. This value can be passed to `fseek()` as an input parameter.

For a binary stream, the position indicator is the number of characters from the beginning of the file. For a text stream, the position indicator contains unspecified information, usable by `fseek` function.

If `stream` refers to an I/O device, this function does nothing and returns a zero.

Arguments

`stream` Points to an open pREPC+ stream.

Return Value

If successful, this function returns the current file position indicator. It returns EOF if an error occurs, and sets the `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fseek, *fsetpos*, *fgetpos*

fwrite

Writes to a stream.

```
#include <stdarg.h>
#include <stdio.h>
size_t fwrite(
   const void *ptr,    /* output buffer */
   size_t size,        /* item size */
   size_t nmemb,       /* item count */
   FILE *stream        /* stream pointer */
   )
```


Description

The `fwrite()` function writes, from the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size` to the stream pointed to by `stream`.

The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Arguments

`ptr`	Points to the output buffer.
`size`	Specifies the size of each item to write.
`nmemb`	Specifies the number of items to write.
`stream`	Points to an open pREPC+ stream.

Return Value

This function returns the number of items written. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fread

getc

Gets a character from a stream.

```
#include <stdarg.h>
#include <stdio.h>
int getc(
   FILE *stream   /* stream pointer */
   )
```

Description

This function is equivalent to the `fgetc()` function. It reads the next character from a specified stream.

Arguments

`stream`	Points to an open pREPC+ stream.

Return Value

This function returns the character that is read. If an end-of-file condition is detected, the stream's end-of-file indicator is set. If a read error occurs, the stream's error flag is set. In both cases, `EOF` is returned. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

putc

getchar

Gets a character from stdin.

```
#include <stdarg.h>
#include <stdio.h>
int getchar(
   void
   )
```

Description

The getchar() function reads the next character from the standard input device. It is equivalent to getc(stdin).

Return Value

This function returns the character that is read. If EOF is detected, the stdin EOF flag is set. If a read error occurs, stdin's error flag is set. In both cases, EOF is returned. If an error occurs, errno is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

putchar, *getc*

gets

Gets a string from `stdin`.

```
#include <stdarg.h>
#include <stdio.h>
char *gets(
   char *s   /* buffer */
   )
```

Description

The `gets()` function reads characters from the standard input device into a user buffer(`s`). It continues to read characters until a new-line character is read or an end-of-file condition is detected. Any new-line character is discarded, and a null character is added after the last character read into the user buffer.

Arguments

`s`	Points to the user buffer.

Return Value

If successful, the `gets()` function returns `s`. If a read error occurs or an end-of-file condition is detected before any characters are read, a null pointer is returned. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = YES`

See Also

puts, *fgets*

gmtime

Converts the calendar time to broken-down time.

```
#include <time.h>
struct tm *gmtime (
   const time_t *timer    /* calendar time */
   )
```

Description

The `gmtime()` function converts the calendar time pointed to by timer into broken-down time, expressed as Coordinated Universal Time (UTC).

The calendar time is generally obtained through a call to `time()`.

Arguments

`timer` Points to the calendar time.

Return Value

The `gmtime()` function always returns NULL, since the concept of UTC is not supported by pREPC+.

Error Codes

None.

Notes

This function is non-reentrant as it returns a pointer to a statically allocated data area. It is provided only to meet the ANSI requirement. The reentrant version of this function is `localtime_r()`.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

gmtime_r, *time*, *localtime*, *localtime_r*, *mktime*

gmtime_r (Reentrant) Converts the calendar time to broken-down time.

```
#include <time.h>
struct tm *gmtime_r (
   const time_t *timer,    /* calendar time */
   struct tm *resultp      /* result */
   )
```

Description

`gmtime_r()` is the reentrant version of the ANSI function `gmtime()`, as defined by POSIX 1003.1c. It converts the calendar time pointed to by `timep` into broken-down time, expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure pointed to by `resultp`.

The calendar time is generally obtained through a call to `time()`.

Arguments

`timer`	Points to the calendar time.
`resultp`	Points to the structure of type `tm` where `gmtime_r()` stores the result. The `tm` structure is defined in the `mktime()` description on page 4-159.

Return Value

`gmtime_r()` always returns NULL, since the concept of UTC is not supported by pREPC+.

Error Codes

No error codes are returned.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

gmtime, *time*, *localtime*, *localtime_r*, *mktime*

isalnum

Tests for an alphanumeric character.

```
#include <ctype.h>
int isalnum(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an alphanumeric character. An alphanumeric character is a character for which either `isdigit` or `isalpha` is true.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isalnum` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isalnum
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isalpha, *isdigit*

isalpha Tests for an alphabetic character.

```
#include <ctype.h>
int isalpha(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an uppercase or lowercase letter.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isalpha` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isalpha
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

iscntrl Tests for a control character.

```
#include <ctype.h>
int iscntrl(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an ASCII control character. ASCII control characters are those whose values lie between 0 and 31, inclusive, and the character 127 (DEL).

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `iscntrl` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef iscntrl
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isprint

isdigit Tests for a digit.

```
#include <ctype.h>
int isdigit(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is a decimal digit (0 through 9).

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isdigit` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isdigit
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

isgraph Tests for a graphical character.

```
#include <ctype.h>
int isgraph(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an ASCII graphical character. ASCII graphical characters are those whose values lie from 33 (exclamation point) through 126 (tilde). This includes all of the printable characters except the space (‘ ‘).

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isgraph` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isgraph
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isprint

islower Tests for a lowercase letter.

```
#include <ctype.h>
int islower(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is a lowercase letter.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `islower` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef islower
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isupper, *isalpha*

isprint

Tests for a printable character.

```
#include <ctype.h>
int isprint(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an ASCII printable character. An ASCII printable character is any character that is not a control character. Their values lie between 32 (space) and 126 (tilde), inclusive.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isprint` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isprint
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isgraph, *iscntrl*

ispunct Tests for a punctuation character.

```
#include <ctype.h>
int ispunct(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an ASCII punctuation mark character. An ASCII punctuation mark character is any printing character that is neither a space nor a character for which `isalnum` is true.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `ispunct` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef ispunct
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isalnum, *isprint*, *isalpha*, *isdigit*

isspace Tests for a space.

```
#include <ctype.h>
int isspace(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is a tab ('`\t`'), line-feed ('`\n`'), vertical tab ('`\v`'), form-feed ('`\f`'), carriage return ('`\r`'), or space character (' ').

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isspace` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isspace
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

isupper Tests for an uppercase letter.

```
#include <ctype.h>
int isupper(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is an uppercase letter.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isupper` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isupper
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isalpha, *islower*

isxdigit Tests for a hexadecimal digit.

```
#include <ctype.h>
int isxdigit(
   int c   /* character */
   )
```

Description

This function tests the value in `c` (converted to an `unsigned char`) to see if it is a hexadecimal digit. The hexadecimal digits are defined as the ASCII representation of digits `0` through `9`, lower case letters `a` through `f` and uppercase letters `A` through `F`.

Arguments

`c`	Specifies the value to be tested.

Return Value

This function returns a nonzero value if the test result is true. It returns a 0 if the result is false.

Error Codes

Refer to Appendix A.

Notes

The `isxdigit` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef isxdigit
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

isdigit

labs

Computes the absolute value of a long integer.

```
#include <stdlib.h>
long labs (
   long j   /* long integer */
   )
```

Description

The `labs()` function converts the long integer `j` into its absolute value. If the result cannot be represented, the behavior is undefined.

Arguments

`j`	Specifies the long integer to be converted.

Return Value

`labs()` returns the absolute value.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

labs

ldexp

Multiplies a floating-point number by an integral power of 2.

```
#include <math.h>
double ldexp (
   double x,
   int exp
   )
```

Description

The ldexp function multiplies a floating-point number by an integral power of 2. A range error can occur.

Arguments

x	The multiplicand.
exp	The integral power of 2 by which to multiply.

Return Value

ldexp returns the value of x times 2 raised to the power exp.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

frexp

ldiv

Performs a division operation on two specified long integers.

```
#include <stdlib.h>
ldiv_t ldiv (
   long numer,    /* numerator */
   long denom     /* denominator */
   )
```

Description

The `ldiv()` function computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, `quot * denom + rem` is equal to `numer`.

Arguments

`numer`	Specifies the numerator.
`denom`	Specifies the denominator.

Return Value

The `ldiv()` function returns a structure of type `ldiv_t`. This structure is defined in `stdlib.h` as follows:

```
typedef struct {
   long quot;   /* the quotient */
   long rem;    /* the remainder */
   } ldiv_t;
```

Error Codes

None.

Notes

Callable From

- Task

- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

div

localeconv Obtains the current locale settings.

```
#include <locale.h>
struct lconv *localeconv(void)
```

Description

The localeconv() function obtains the current locale settings that relate to numeric values, putting them into a statically allocated structure of type lconv. It returns a pointer to that structure.

The members of lconv with type char * are pointers to strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type char are nonnegative numbers, any of which can be CHAR_MAX to indicate the value is not available in the current locale.

The lconv structure is defined as follows:

```
struct lconv {
   char *decimal_point;     /* Decimal point character for
                             * non-monetary values */
   char *thousands_sep;     /* Thousands separator for
                             * non-monetary values */
   char *grouping;          /* Specifies grouping for
                             * non-monetary values */
   char *int_curr_symbol;   /* International currency symbol */
   char *currency_symbol;   /* The local currency symbol */
   char *mon_decimal_point; /* Decimal point character for
                             * monetary values */
   char *mon_thousands_sep; /* Thousands separator for
                             * monetary values */
   char *mon_grouping;      /* Specifies grouping for
                             * monetary values */
   char *positive_sign;     /* Positive value indicator
                             * for monetary values */
   char *negative_sign;     /* Negative value indicator
                             * for monetary values */
   char int_frac_digits;    /* Number of digits displayed
                             * to the right of the decimal
                             * point for monetary values
                             * displayed using international
                             * format */
   char frac_digits;        /* Number of digits displayed
                             * to the right of the decimal
                             * point for monetary values */
```

```
    char p_cs_precedes;      /* 1 if currency symbol precedes
                              * positive value, 0 if currency
                              * symbol follows value */
    char p_sep_by_space;     /* 1 if currency symbol is
                              * separated from value by a
                              * space, 0 otherwise */
    char n_cs_precedes;      /* 1 if currency symbol precedes
                              * a negative value, 0 if currency
                              * symbol follows value */
    char n_sep_by_space;     /* 1 if currency symbol is
                              * separated from a negative value
                              * by a space, 0 if currency symbol
                              * follows value */
    char p_sign_posn;        /* Indicates position of positive
                              * value symbol */
    char n_sign_posn;        /* Indicates position of negative
                              * value symbol */
};
```

The elements of `grouping` and `mon_grouping` are interpreted according to the following:

`CHAR_MAX`	No further grouping is to be performed.
`0`	The previous element is to be repeatedly used for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

`0`	Parentheses surround the quantity and `currency_symbol`.
`1`	The sign string precedes the quantity and `currency_symbol`.
`2`	The sign string succeeds the quantity and `currency_symbol`.
`3`	The sign string immediately precedes `currency_symbol`.
`4`	The sign string immediately precedes `currency_symbol`.

Return Value

This function returns a pointer to the filled-in `lconv` structure. This structure must not be changed by your program, but it may be overwritten by a subsequent call to

`localeconv()`. In addition, calls to `setlocale()` with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`setlocale`

localtime

Converts the calendar time to broken-down time.

```
#include <time.h>
struct tm *localtime (
   const time_t *timer    /* calendar time */
   )
```

Description

The `localtime()` function converts the calendar time pointed to by `timer` into broken-down time. The time is represented in local time.

The calendar time is generally obtained through a call to `time()`.

Arguments

`timer` Points to the calendar time.

Return Value

The `localtime()` function returns a pointer to the `tm` structure that contains the broken-down time. The `tm` structure is defined in the `mktime()` description on page 4-159.

Error Codes

None.

Notes

This function is non-reentrant as it returns a pointer to a statically allocated data area. It is provided only to meet the ANSI requirement. The reentrant version of this function is `localtime_r()`.

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

localtime_r, *time*, *gmtime*, *gmtime_r*, *mktime*

localtime_r (Reentrant) Converts the calendar time to broken-down time.

```
#include <time.h>
struct tm *localtime_r (
   const time_t *timer,   /* pointer to calendar time */
   struct tm *resultp     /* pointer to result */
   )
```

Description

`localtime_r()` is the reentrant version of the ANSI function `localtime()`, as defined by POSIX 1003.1c. It converts the calendar time pointed to by `timep` into broken-down time and stores it in the structure pointed to by `resultp`. The time is represented in local time.

The calendar time is generally obtained through a call to `time()`.

Arguments

`timer`	Points to the calendar time.
`resultp`	Points to the `tm` structure where `localtime_r()` stores the result. The `tm` structure is defined in the `mktime()` description on page 4-159.

Return Value

Upon success, `localtime_r()` returns the value of `resultp`. On failure, it returns NULL; the only cause of failure is if `timer` points to a negative value.

Error Codes

No error codes are returned.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = YES
```

See Also

localtime, *time*, *gmtime*, *gmtime_r*, *mktime*

log

Computes the natural logarithm of x.

```
#include <math.h>
double log (
   double x
   )
```

Description

The log function computes the natural logarithm of x.

Arguments

x	The number whose logarithm is to be computed. It must be a positive, non-zero number.

Return Value

log returns the natural logarithm of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

exp, *log10*

log10 Computes the base-ten logarithm of x.

```
#include <math.h>
double log10 (
   double x
   )
```

Description

The `log` function computes the base-ten logarithm of x.

Arguments

x	The number whose base-ten logarithm is to be computed. It must be a positive, non-zero number.

Return Value

`log10` returns the base-ten logarithm of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

log, *exp*

longjmp

Restores the environment set by the most recent invocation of setjmp.

```
#include <setjmp.h>
void longjmp (
   jmp_buf env,
   int val
   )
```


Description

The longjmp function restores the environment saved by the most recent invocation of the setjmp macro in the same invocation of the program, with the corresponding jmp_buf argument. If setjmp has not been invoked, or if the setjmp macro finished execution after longjmp was invoked, there is no way of telling how longjmp will act.

All accessible objects have values as of the time longjmp was called, except that the values of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the setjmp invocation and longjmp call are indeterminate.

As it bypasses the usual function call and return mechanisms, the longjmp function executes correctly in contexts of interrupts and any of their associated functions. However, if the longjmp function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

Arguments

jmp_buf	The argument of setjmp to which the environment is to be restored.
val	The value to be used as the return value of the setjmp macro.

Return Value

Program execution continues as if the setjmp macro had just returned the value set by val. The longjmp function cannot cause the setjmp macro to return the value zero; if val is zero, the setjmp macro returns the value one.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

setjmp

malloc Allocates memory.

```
#include <stdlib.h>
void *malloc(
   size_t size    /* element size */
   )
```

Description

The `malloc()` function allocates memory for an object whose size is specified by `size` and whose value is indeterminate. `malloc()` calls the pSOS+ region manager to allocate the memory. The caller can be blocked if memory is not available and the wait option is selected in the pREPC+ Configuration Table. Memory is allocated from Region 0.

Arguments

`size` Specifies the number of bytes to allocate.

Return Value

This function returns either a pointer to the allocated memory or a null pointer if no memory is allocated. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

mblen

Determines the number of bytes in a multibyte character.

```
#include <stdlib.h>
int mblen (
   const char *s,    /* character */
   size_t n          /* size of character */
   )
```

Description

If s is not a null pointer, the mblen() function determines the number of bytes contained in the multibyte character pointed to by s. Except that the shift state of the mbtowc() function is not affected, it is equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

Arguments

s	Points to the character to be examined.
n	Specifies the size of s.

Return Value

If s is a null pointer, this function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, this function either returns 0 (if s points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

mbtowc

mbstowcs

Converts a multibyte character string into a wide character string.

```
#include <stdlib.h>
size_t mbstowcs (
   wchar_t *pwcs,   /* wide string */
   const char *s,   /* original multibyte string */
   size_t n         /* wide string length */
   )
```

Description

The `mbstowcs()` function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by `s` into a sequence of corresponding codes and stores not more than `n` codes into the array pointed to by `pwcs`. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the `mbtowc()` function, except that the shift state of the `mbtowc()` function is not affected.

If copying takes place between objects that overlap, the behavior is undefined.

Arguments

`pwcs`	Points to the array where `mbstowcs()` stores the converted string.
`s`	Points to the string to be converted.
`n`	Specifies the length of `pwcs`.

Return Value

If an invalid multibyte character is encountered, `mbstowcs()` returns `(size_t)-1`. Otherwise, it returns the number of array elements modified, not including a terminating zero code, if any.

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

mbtowc, *wcstombs*, *wctomb*

mbtowc

Converts a multibyte character into its wide character equivalent.

```
#include <stdlib.h>
int mbtowc (
   wchar_t *pwc,    /* result wide character */
   const char *s,   /* original multibyte character */
   size_t n         /* size of original character */
   )
```

Description

If s is not a null pointer, the mbtowc() function determines the number of bytes that are contained in the multibyte character pointed to by s. It then determines the code for the value of type wchar_t that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and pwc is not a null pointer, the mbtowc() function stores the code in the object pointed to by pwc. At most n bytes of the array pointed to by s will be examined.

Arguments

pwc	Points to the array where mbtowc() stores the result character.
s	Points to the multibyte character to be converted.
n	Specifies the size of the multibyte character to be converted.

Return Value

If s is a null pointer, this function returns a nonzero or zero value, if multibyte codings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, this function either returns 0 (if s points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than n or the value of the MB_CUR_MAX macro.

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

mbstowcs, *wctomb*, *wcstombs*

memccpy

Copies characters from one memory area to another.

```
#include <memory.h>
   char *memccpy(
   char *s1,          /* destination address */
   char *s2,          /* source address */
   int c,             /* character to check for */
   int n              /* length of copy */
   )
```

Description

This function copies characters from memory areas `s2` into `s1`, stopping after the first occurrence of character `c` has been copied, or after `n` characters have been copied, whichever comes first.

Arguments

`s1`	Points to the destination object.
`s2`	Points to the source object.
`c`	Specifies the character for which to check for early termination of the copy.
`n`	Specifies the number of characters to be copied.

Return Value

This function returns the pointer to the character after the copy of `c` in `s1`, or a NULL pointer if `c` was not found in the first `n` characters of `s2`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

memcpy

memchr

Searches memory for a character.

```
#include <string.h>
void *memchr(
   const void *s,   /* target buffer */
   int c,           /* character key */
   size_t n         /* search length */
   )
```

Description

This function searches for the first occurrence of the character `c` (converted to an `unsigned char`) in the first `n` characters of the object pointed to by `s`.

Arguments

`s`	Points to the buffer to be searched.
`c`	Specifies the character to be searched for.
`n`	Specifies the number of characters to search through.

Return Value

This function returns a pointer to the located character, or a null pointer if the character is not found.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

strchr

memcmp Compares two objects in memory.

```
#include <string.h>
int memcmp(
   const void *s1,   /* buffer 1 */
   const void *s2,   /* buffer 2 */
   size_t n          /* comparison length */
   )
```

Description

This function compares `n` characters in the buffers pointed to by `s1` and `s2`.

Arguments

`s1`	Points to the first buffer.
`s2`	Points to the second buffer.
`n`	Specifies the number of characters to be compared.

Return Value

This function returns a value that is either greater than, equal to, or less than zero. The result depends on whether the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

strcmp

memcpy Copies characters in memory.

```
#include <string.h>
void *memcpy(
   void *s1,          /* destination address */
   const void *s2,    /* source address */
   size_t n           /* source length */
   )
```

Description

This function copies n characters from the object pointed to by s2 into the object pointed to by s1. If the memory areas overlap, the result is unpredictable.

Arguments

s1	Points to the destination object.
s2	Points to the source object.
n	Specifies the number of characters to be copied.

Return Value

This function returns the value of s1.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

memccpy, *memmove*, *strcpy*, *strncpy*

memmove Copies characters in memory.

```
#include <string.h>
void *memmove(
   void *s1,          /* destination address */
   const void *s2,    /* source address */
   size_t n           /* source length */
   )
```

Description

This function copies a specified number of characters from one object into another. `memmove()` copies the data correctly even if the memory areas overlap (unlike `memcpy()`).

Arguments

`s1`	Points to the source object.
`s2`	Points to the destination object.
`n`	Specifies the number of characters to be copied.

Return Value

This function returns the value of `s1`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

memcpy, *strcpy*, *strncpy*

memset

Initializes memory with a given value.

```
#include <string.h>
void *memset(
   void *s,    /* object address */
   int c,      /* initialization value */
   size_t n    /* number of characters */
   )
```

Description

The `memset()` function copies the value of `c` (converted to an `unsigned char`) into each of the first `n` characters in the object pointed to by `s`.

Arguments

`s`	Points to the object where the character is to be copied.
`c`	Specifies the value to be copied.
`n`	Specifies the number of characters in the object to be initialized.

Return Value

The function returns the value of `s`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

mktime

Converts the broken-down time into calendar time.

```
#include <time.h>
time_t mktime (
   struct tm *timeptr    /* broken-down time */
   )
```

Description

The `mktime()` function converts the broken-down time, expressed as local time, in the structure pointed to by `timeptr` into calendar time with the same encoding as the `time()` function. This function is primarily used to initialize the system time. The elements `tm_wday` and `tm_yday` are set by the function, so they need not be defined prior to the call. The original values of the other components are not restricted to the normal ranges (see below).

Upon successful completion, the values of `tm_wday` and `tm_yday` are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the normal ranges; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

Arguments

`timeptr` Points to a structure of type `tm` that stores the broken-down time. The `tm` structure is defined as follows. The semantics of the members and their normal ranges are expressed in the comments.

```
struct tm {
   int tm_sec;    /* seconds after the minute [0,61] */
   int tm_min;    /* minutes after the hour [0,59] */
   int tm_hour;   /* hours since midnight [0,23] */
   int tm_mday;   /* day of the month [1,31] */
   int tm_mon;    /* months since January [0,11] */
   int tm_year;   /* years since 1900 */
   int tm_wday;   /* days since Sunday [0,6] */
   int tm_yday;   /* days since January 1 [0,365] */
   int tm_isdst;  /* Daylight Saving Time flag */
   };
```

The range [0, 61] for `tm_sec` allows for up to two leap seconds. The value of `tm_isdst` is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

Return Value

The `mktime()` function returns the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t) -1`.

Error Codes

None.

Example

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
   “Sunday”, “Monday”, “Tuesday”, “Wednesday”,
   “Thursday”, “Friday”, “Saturday”, “-unknown-”
   };
struct tm time_str;

/* ... */

time_str.tm_year = 2001 - 1900;
time_str.tm_mon = 7 - 1;
time_str.tm_mday = 4;
time_str.tm_hour = 0;
time_str.tm_min = 0;
time_str.tm_sec = 1;
time_str.tm_isdst = -1;

if (mktime(&time_str) == -1)
   time_str.tm_wday = 7;

printf(“%s\n”, wday[time_str.tm_wday]);
```

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = YES
```

See Also

time, *localtime*, *localtime_r*

modf

Breaks the argument value into integral and fractional parts.

```
#include <math.h>
double modf (
   double value,
   double *iptr
   )
```

Description

The `modf` function breaks the argument value into integral and fractional parts, each which has the same sign as the argument. `modf` stores the integral part as a double in the object pointed to by `iptr`.

Arguments

`value`	Value to be broken into parts.
`lptr`	Pointer to a double where the integral part of the result is stored.

Return Value

`modf` returns the signed fractional part of `value`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

fmod

perror Prints a diagnostic message.

```
#include <stdarg.h>
#include <stdio.h>
void perror(
   const char *s   /* error string */
   )
```

Description

The `perror()` function writes the string pointed to by `s` followed by a diagnostic message to the standard error device. The diagnostic message is a function of the calling task's `errno` value.

Arguments

`s`	Points to the string to write (error message).

Return Value

This function does not return a value.

Error Codes

No error codes are returned.

Notes

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`errno`

pow

Computes x raised to the power y.

```
#include <math.h>
double pow (
   double x,
   double y
   )
```

Description

The pow function computes x raised to the power y. A domain error occurs if:

- x is negative and y is not an integral value, or
- the result cannot be represented when x is zero and y is less than or equal to zero.

A range error can occur.

Arguments

x	The number whose power is to be computed.
y	The power.

Return Value

pow returns the value of x raised to the power y.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

None.

printf

Prints formatted output to stdout.

```
#include <stdarg.h>
#include <stdio.h>
long printf(
   const char *format,   /* format control */
   ...                   /* arguments 1 through n */
   )
```


Description

The printf() function is equivalent to fprintf(), except that the output is directed to the standard output device instead of a file designated by an input parameter.

Arguments

format	Points to the format control string. For more information, see fprintf on page 4-66.
...	Arguments 1 through n to be written according the specifications of the control string.

Return Value

This function returns either the number of characters written or EOF if a write error occurs. If an error occurs, errno is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

scanf, *fprintf*

putc

Writes a character to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int putc(
   int c,          /* character */
   FILE *stream    /* stream pointer */
   )
```


Description

The `putc()` function writes the character specified by `c` (converted to an unsigned `char`) to the stream pointed to by `stream`. `stream`'s position indicator (if defined) is advanced on a successful write.

Arguments

`c`	Specifies the character to write.
`stream`	Points to an open pREPC+ stream.

Return Value

The `putc()` function returns `c`. If a write error occurs, the error flag for the stream is set, `EOF` is returned, and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

getc, *putchar*

putchar

Writes a character to stdout.

```
#include <stdarg.h>
#include <stdio.h>
int putchar(
   int c   /* character */
   )
```

Description

The putchar() function writes the character c to the standard output stream after converting it to an unsigned char.

Arguments

c	Specifies the character to write.

Return Value

The putchar() function returns letter. If a write error occurs, the error flag for the standard output stream is set, EOF is returned, and errno is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

getchar, *putc*

puts

Writes a string to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int puts(
   const char *s   /* string */
   )
```

Description

The `puts()` function writes a string to the standard output stream, and appends a new-line character to the output. The terminating null character is not written.

Arguments

`s`	Points to the string to write.

Return Value

The `puts()` function returns 0 if the operation is successful and `EOF` if the operation fails. If an error occurs, the stream's error indicator and `errno` are set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

gets, *fputs*

qsort Sorts an array.

```
#include <stdlib.h>
void qsort(
   void *base,    /* array base */
   size_t nmemb,  /* array length */
   size_t size,   /* array element size */
   int (*compar) (const void *, const void *)
                  /* comparison function */
   )
```


Description

The `qsort()` function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.

The array is sorted in ascending order according to the user-supplied function pointed to by `compar`. The `compar` function is called with two arguments that point to the objects being compared. The `compar` function must return an integer that is less than, equal to, or greater than 0 if the first argument is considered less than, equal to, or greater than the second argument, respectively. The `compar` function can call all pREPC+ character handling functions and all pREPC+ string handling functions except `strtok()`. Other pREPC+ functions cannot be called from `compar`.

If two members of the array are equal, their order in the sorted array is unspecified.

Arguments

`base`	Points to the beginning of the array.
`nmemb`	Specifies the length of the array.
`size`	Specifies the size of each member of the array.
`compar`	Points to the user-supplied comparison function.

Return Value

This function does not return a value.

Error Codes

None.

Notes

Callable From

- Task
- ISR, provided the `compar` function can be called from an ISR.

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`bsearch`

rand

Returns a pseudo-random number.

```
#include <stdlib.h>
int rand(
   void
   )
```

Description

The `rand()` function generates a pseudo-random integer number in the range 0 through 32,767. This function works in concert with `srand()`.

Return Value

This function returns the random number generated.

Error Codes

None.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

rand_r, *srand*

rand_r

Returns a pseudo-random sequence of integers with repeated calls.

```
#include <stdlib.h>
int *rand_r(
   unsigned int *seed         /* seed for random generator */
   )
```

Description

The function rand_r takes an input seed value (pointed to by the argument seed) and computes a pseudo-random integer in the range 0 to RAND_MAX (which must be at least 32767), and also computes a successor seed value.

By using the successor seed value from the previous rand_r call, repeated calls to rand_r may be used to generate a sequence of pseudo-random numbers. The same input seed always returns the same random number and the same successor seed. Thus, a sequence of rand_r calls, using a particular initial seed, will always generate the same sequence of pseudo-random numbers.

Arguments

seed	A pointer to a memory location that contains the input seed value and that receives the successor seed value.

Return Value

Each call of this function returns a pseudo-random integer, and also returns a successor seed value.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

rand, *srand*

realloc Allocates memory.

```
#include <stdib.h>
void *realloc(
   void *ptr,   /* pointer */
   size_t size  /* new size */
   )
```

Description

The `realloc()` function changes the size of the object pointed to by `ptr` to the size specified by `size`. The contents of the object remain unchanged up to the lesser of the new or old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate.

The caller can be blocked if memory is not available and the wait option is selected in the pREPC+ Configuration Table.

Arguments

`ptr`	Points to the object whose size is to be changed. If `ptr` is a null pointer, `realloc()` behaves like `malloc()`. If `ptr` does not match a segment previously returned by `calloc()`, `malloc()` or `realloc()`, the result is unpredictable. If `ptr` is not a null pointer and `size` is 0, the segment is deallocated.
`size`	Specifies the new size of the object.

Return Value

This function returns a pointer to the possibly moved allocated memory or a null pointer. If an error occurs, `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

calloc, *malloc*, *realloc*

remove Removes a file.

```
#include <stdarg.h>
#include <stdio.h>
int remove(
   const char *filename   /* file name */
   )
```

Description

This function removes the file whose name is pointed to by `filename`. Before a file can be removed, it must be closed. This restriction does not apply to files residing on NFS volumes.

Arguments

`filename` Points to the string that contains the filename.

Return Value

This function returns zero if the file is removed. If the file is not removed or if `filename` specifies an I/O device, then a nonzero value is returned and `errno` is set.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

rename Renames a file.

```
#include <stdarg.h>
#include <stdio.h>
int rename(
   const char *old,    /* existing file name */
   const char *new     /* new file name */
   )
```


Description

This function changes the name of a file from old to new. It applies only to pHILE+-managed files. If a file new already exists, it is deleted. Certain error conditions specific to the volume type can result, as follows:

- If a file called new exists and is open, the call fails (except on NFS volumes).
- If old is open, the call fails (DOS volumes only).
- If old is a directory file, the call fails (DOS volumes only).

Arguments

old	Points to the string that contains the old filename.
new	Points to the string that contains the new filename.

Return Value

This function returns zero if the file is successfully renamed. If an error occurs, errno is set, and a nonzero value is returned.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

```
SC_PREPC = YES
```

rewind

Resets the file position indicator.

```
#include <stdarg.h>
#include <stdio.h>
void rewind(
   FILE *stream   /* stream pointer */
   )
```

Description

The `rewind()` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that the stream's error indicator is cleared.

Arguments

`stream` Points to an open pREPC+ stream.

If `stream` refers to an I/O device, this function does nothing and returns a 0.

Return Value

This function does not return a value.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

ftell, *fgetpos*, *fsetpos*, *fseek*

scanf

Reads formatted input from `stdin`.

```
#include <stdarg.h>
#include <stdio.h>
int scanf(
   const char *format,   /* format control */
   ...                   /* arguments 1 through n */
   )
```


Description

The `scanf()` function is equivalent to `fscanf()` except that `scanf()` takes the input from the standard input device.

Arguments

`format`	Points to the format control string. For more information, see `fscanf` on page 4-83.
`...`	Arguments 1 through n point to variables where input is stored.

Return Value

This function returns `EOF` and sets `errno` if an input failure occurs before any conversion. Otherwise, it returns the number of input items assigned, which can be fewer than provided, even zero, in the event of an early matching failure.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

printf, *scanf*, *sscanf*

setbuf

Changes a stream's buffer.

```
#include <stdarg.h>
#include <stdio.h>
void setbuf(
   FILE *stream,   /* stream pointer */
   char *buf       /* I/O buffer */
   )
```


Description

The `setbuf()` function causes a new buffer to be used for a specified stream (instead of the buffer that is currently assigned).

The buffer is assumed to have a size equal to `lc_bufsize`, which is defined in the pREPC+ Configuration Table.

The `setbuf()` function must be called after the specified stream has been opened but prior to performing any read or write operation on the stream. If `setbuf()` is called after a read or write operation, it has no effect.

You must use the `setvbuf()` call if you want additional control over buffering options.

Arguments

`stream`	Points to an open pREPC+ stream.
`buf`	Points to the new buffer. If `buf` is a null pointer, all input and output is unbuffered. This effectively eliminates buffering. Otherwise, the stream is set to fully buffered mode.

Return Value

This function does not return a value.

Notes

Callable From

- Task

Error Codes

Refer to Appendix A.

Required SC_PREPC Setting

SC_PREPC = YES

See Also

setvbuf

setjmp

Saves the calling environment for later restoration.

```
#include <setjmp.h>
int setjmp (
   jmp_buf env
   )
```

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later restoration by the `longjmp` function.

Arguments

`jmp_buf` The environment to be saved.

Return Value

If the return is from a direct invocation, the `set_jmp` macro returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` macro returns a nonzero value.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

longjmp

setlocale

Obtains or changes the program's locale.

```
#include <locale.h>
char *setlocale (
   int category          /* localization category */
   const char *locale    /* locale */
   )
```

Description

The `setlocale()` function allows you to query or set certain parameters that are sensitive to the geo-political location where a program is used. For example, in Europe, the comma is sometimes used in place of the decimal point.

To query the locale or a portion thereof, you set `locale` to point to NULL. To change the locale, or a portion thereof, you set `locale` to point to a string that specifies the desired value.

Arguments

`category` Specifies the localization category to be queried or changed, and must be one of the following:

`LC_ALL`	All categories.
`LC_COLLATE`	Affects the behavior of `strcoll()` and `strxfrm()`.
`LC_CTYPE`	Affects the behavior of the character-handling functions (`isalnum()`, etc.) and the multibyte functions.
`LC_MONETARY`	Affects the monetary formatting information returned by `localeconv()`.
`LC_NUMERIC`	Affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by `localeconv()`.
`LC_TIME`	Affects the behavior of `strftime()`.

`locale` Points to a string specifying the locale in which the program is used. Only one value is supported by pREPC+:

"C" Specifies the minimal environment for C translation.

Return Value

If a pointer to a string is given for `locale` and the selection can be honored, `setlocale()` returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, `setlocale()` returns a null pointer and the program's locale is not changed.

A null pointer for `locale` causes `setlocale()` to return a pointer to the string associated with the `category` for the program's current locale; the program's locale is not changed.

The pointer to string returned by `setlocale()` is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to will not be modified by the program, but may be overwritten by a subsequent call to `setlocale()`.

Error Codes

None.

Notes

pREPC+ supports only the "C" locale defined by ANSI. The "C" locale does not support multi-byte encoding; under the "C" locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = YES

See Also

localeconv, *strcoll*, *strxfrm*, *strftime*

setvbuf

Changes a stream's buffering characteristics.

```
#include <stdarg.h>
#include <stdio.h>
int setvbuf(
   FILE *stream,  /* stream pointer */
   char *buf,     /* I/O buffer */
   int mode,      /* access mode */
   size_t size    /* buffer size */
   )
```

Description

The `setvbuf()` function can be used to change either the buffer associated with a stream, the size of a stream's buffer, or the method employed for buffering the stream's data.

`setvbuf()` must be called after the specified stream has been opened, but prior to reading or writing the file. If `setbuf()` is called after a read or write operation, it has no effect.

pREPC+ allocates a buffer automatically if the `buf` parameter is NULL and a non-zero `size` is specified.

Arguments

`stream` Points to an open pREPC+ stream.

`buf` Points to the new buffer to be associated with the stream.

`mode` Defines the method employed for buffering data. The possible modes are as follows:

`_IO_FBF`	Input/output is fully buffered.
`_IO_LBF`	Input/output is line buffered.
`_IO_NBF`	Input/output is not buffered.

If a stream is fully buffered, it is flushed only when it is full. If it is line buffered, the buffer is flushed either when it is full or a when a new-line character is detected. `_IO_NBF` eliminates buffering; it is functionally equivalent to defining `size` equal to 0 and `buf` equal to NULL.

`size` Specifies the size of the buffer. `size` overrides the default buffer size defined in the pREPC+ Configuration Table.

Return Value

This function returns a 0 if it succeeds and a negative number if it fails.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

setbuf

sin

Computes the sine of x (measured in radians).

```
#include <math.h>
double sin (
   double x
   )
```

Description

The `sin` function computes the sine of x (measured in radians).

Arguments

x	The angle whose sine is to be computed.

Return Value

`sin` returns the sine of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

asin, *sinh*

sinh

Computes the hyperbolic sine of x.

```
#include <math.h>
double sinh (
   double x
   )
```

Description

The `sinh` function computes the hyperbolic value of the sine of x. A range error occurs if the magnitude of x is too large.

Arguments

x	The angle whose hyperbolic sine is to be computed.

Return Value

`sin` returns the hyperbolic sine of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

sin, *asin*

sprintf

Writes formatted output to a buffer.

```
#include <stdarg.h>
#include <stdio.h>
int sprintf(
   char *s,              /* buffer */
   const char *format,   /* format control */
   ...                   /* arguments 1 through n */
   )
```

Description

The `sprintf()` function is equivalent to `fprintf()` except that `sprintf()` directs the output to a buffer pointed to by `s`.

Arguments

`s`	Points to the buffer where output is directed.
`format`	Points to the format control string. For more information, see `fprintf` on page 4-66.
`...`	Arguments 1 through n are written by `sprintf()` according to the specifications of the format control string.

Return Value

This function returns the number of characters written in the buffer, not counting the terminating null character.

Error Codes

None.

Notes

Callable From

- Task

Required SC_PREPC Setting

```
SC_PREPC = YES
```

See Also

fprintf, *printf*, *vsprintf*

sqrt Computes the nonnegative square root of x.

```
#include <math.h>
double sqrt (
   double x
   )
```

Description

The `sqrt` function computes the nonnegative square root of `x`. A domain error occurs if the argument is negative.

Arguments

`x`	The number whose square root is to be computed.

Return Value

`sqrt` returns the value of the square root of the given number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

None.

srand

Sets the seed for the random number generator (rand).

```
#include <stdlib.h>
void srand(
   unsigned int seed   /* seed value */
   )
```

Description

This function works in concert with rand(). The srand() function uses seed as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand().

For a given seed value, the sequence of random numbers generated is the same. By default, a sequence of random numbers is generated using a seed value of 1.

Arguments

seed	Specifies the seed value.

Return Value

This function does not return a value.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

rand

sscanf

Reads formatted input from a string.

```
#include <stdarg.h>
#include <stdio.h>
int sscanf(
   const char *s,         /* string */
   const char *format,    /* format control */
   ...                    /* arguments 1 through n */
   )
```


Description

The `sscanf()` function is equivalent to `fscanf()`, except that `sscanf()` takes the input from the string pointed to by `s`.

Arguments

`s`	Points to the string to be read.
`format`	Points to the format control string. For more information, see `fscanf` on page 4-83.
`...`	Arguments 1 through n point to variables where input is stored.

Return Value

This function returns the number of variable assignments. If an error occurs, the function returns `EOF` and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

scanf, *fscanf*

strcat

Appends one string to another string.

```
#include <string.h>
char *strcat(
   char *s1,         /* destination string */
   const char *s2    /* source string */
   )
```

Description

This function appends a copy of one string (s2) to the end of another string (s1). The first character in the source string overwrites the terminating null character in the destination string. If copying takes place between strings that overlap, the behavior is undefined.

Arguments

s1	Points to the destination string.
s2	Points to the source string.

Return Value

This function returns the value of s1.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strchr Searches a string for a character.

```
#include <string.h>
char *strchr(
   const char *s,   /* search string */
   int c            /* reference character */
   )
```

Description

This function searches for the first occurrence of `c` (converted to an `unsigned char`) in the string pointed to by `s`.

Arguments

`s`	Points to the string to be searched.
`c`	Specifies the reference character.

Return Value

This function returns a pointer to the located character. If a character is not found, it returns a null pointer.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

strcmp

Compares two character strings.

```
#include <string.h>
int strcmp(
   const char *s1,   /* candidate string */
   const char *s2    /* candidate string */
   )
```

Description

This function compares two character strings and returns a value that reflects whether the first string (`s1`) is greater than, equal to, or less than the second string (`s2`).

Arguments

`s1`	Points to the first string.
`s2`	Points to the second string.

Return Value

This function returns a value greater than, equal to, or less than 0, depending on whether the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strcoll

Compares two character strings.

```
#include <string.h>
int strcoll (
   const char *s1,   /* candidate string */
   const char *s2    /* candidate string */
   )
```

Description

The `strcoll()` function compares the string pointed to by `s1` to the string pointed to by `s2`. The comparison is performed relative to the current locale. (You can specify the locale by using the `setlocale()` function. See `setlocale()` on page 4-191.)

Arguments

`s1`	Points to the first string.
`s2`	Points to the second string.

Return Value

This function returns a value greater than, equal to, or less than 0, depending on whether the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

See Also

setlocale, *strxfrm*

strcpy

Copies one string to another string.

```
#include <string.h>
char *strcpy(
   char *s1,         /* destination string */
   const char *s2    /* source string */
   )
```

Description

This function copies one string (`s2`) including the null character, into another string (`s1`). `strcpy()` continues to copy characters until it detects a null terminator. If the strings overlap, the result is unpredictable.

Arguments

`s1`	Points to the destination string.
`s2`	Points to the source string.

Return Value

This function returns the value of `s1`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strcspn Calculates the length of a substring.

```
#include <string.h>
size_t strcspn(
   const char *s1,  /* candidate string */
   const char *s2   /* reference string */
   )
```

Description

This function calculates the length of the maximum initial segment of a string (`s1`) which consists entirely of characters *not* in another string (`s2`).

Arguments

`s1`	Points to the string to be examined.
`s2`	Points to the reference string.

Return Value

This function returns the length of the segment.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

strerror

Maps an error number to an error message string.

```
#include <string.h>
char *strerror (
   int errnum    /* error number */
   )
```

Description

The `strerror()` function maps the error number in `errnum` to an error message string.

Arguments

`errnum` Specifies the error number.

Return Value

The `strerror()` function returns a pointer to the string. The array pointed to must not be modified by the program, but may be overwritten by a subsequent call to `strerror()`.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = YES

See Also

perror, *errno*

strftime

Places formatted time and date information into a string.

```
#include <time.h>
size_t strftime (
   char *s,                    /* string */
   size_t maxsize,             /* string length */
   const char *format,         /* format control string */
   const struct tm *timeptr    /* broken-down time */
   )
```

Description

The `strftime()` function places time and date information into the string pointed to by `s` as controlled by the string pointed to by `format`.

`format` contains conversion specifiers and ordinary multibyte characters. All ordinary multibyte characters, including the terminating null character, are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed in the array.

Arguments

`s`	Points to the string where `strftime()` places the formatted information.
`maxsize`	Specifies the length of `s`.
`format`	Contains zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion specifier. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the current locale (see `setlocale()` on page 4-191) and by the values contained in the structure pointed to by `timeptr`.

%a	Replaced by the locale's abbreviated weekday name.
%A	Replaced by the locale's full weekday name.
%b	Replaced by the locale's abbreviated month name.
%B	Replaced by the locale's full month name.

%c	Replaced by the locale's appropriate date and time representation.
%d	Replaced by the day of the month as a decimal number (01-31).
%H	Replaced by the hour (24-hour clock) as a decimal number (00-23).
%I	Replaced by the hour (12-hour clock) as a decimal number (01-12).
%j	Replaced by the day of the year as a decimal (1-366).
%m	Replaced by the month as a decimal number (01-12).
%M	Replaced by the minute as a decimal number (00-59).
%p	Replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%S	Replaced by the second as a decimal number (00-61).
%U	Replaced by the week of the year, where Sunday is the first day of a week (0-52).
%w	Replaced by the weekday as a decimal number (0-6), where Sunday is 0.
%W	Replaced by the week of the year, where Monday is the first day (0-53).
%x	Replaced by the locale's appropriate date representation.
%X	Replaced by the locale's appropriate time representation.
%y	Replaced by the year without century as a decimal number (00-99).
%Y	Replaced by the year with century as a decimal number.
%Z	Replaced by the time zone name.
%%	Replaced by the percent sign.

If a conversion specifier is not one of the above, the behavior is undefined.

`timeptr` Points to the `tm` structure that contains the broken-down time. The `tm` structure is defined in the `mktime()` description on page 4-159.

Return Value

If the total number of resulting characters including the terminating null character is not more than `maxsize`, `strftime()` returns the number of characters placed into the array pointed to by `s`, not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`setlocale`, *`time`*, *`mktime`*, *`localtime`*, *`localtime_r`*

strlen Computes string length.

```
#include <string.h>
size_t strlen(
   const char *s   /* string */
   )
```

Description

The `strlen()` function determines the length of a string (`s`), not including the terminating null character.

Arguments

`s`	Points to the string to be measured.

Return Value

This function returns the computed length.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strncat

Appends characters to a string.

```
#include <string.h>
char *strncat(
   char *s1,          /* destination string */
   const char *s2,    /* source string */
   size_t n           /* source length */
   )
```

Description

This function appends up to n characters from one string (s2) to the end of another string (s1). The first character in the source string overwrites the terminating null character in the destination string. A null character and characters that follow it in the source string are not appended.

The resulting string is always null terminated. If the length of the source string is greater than the number of characters specified in the call, only the specified number of characters (not including the null termination character) is appended.

If copying takes place between objects that overlap, the behavior is undefined.

Arguments

s1	Points to the destination string.
s2	Points to the source string.
n	Specifies the number of characters to append.

Return Value

This function returns the value of s1.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

strncmp

Compares characters in two strings.

```
#include <string.h>
int strncmp(
   const char *s1,   /* first string */
   const char *s2,   /* second string */
   size_t n          /* comparison size */
   )
```

Description

This function compares up to n characters in two strings and returns a value that reflects whether the characters from the first string (s1) are greater than, equal to, or less than the characters from the second string (s2). Characters that follow a null character in the first string are not compared.

Arguments

s1	Points to the first string.
s2	Points to the second string.
n	Specifies the number of characters to compare.

Return Value

This function returns a value greater than, equal to, or less than 0, and the value depends on whether the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

strncpy

Copies characters from one string to another.

```
#include <string.h>
char *strncpy(
   char *s1,          /* destination string */
   const char *s2,    /* source string */
   size_t n           /* source length */
   )
```

Description

This function copies up to `n` characters from one string (`s2`) to another string (`s1`). If the length of the source string is less than the specified number of characters, null characters are copied into the destination string until the specified number have been written. If the length of the source string is greater than or equal to the specified number of characters, no null characters are appended to `s1`.

Arguments

`s1`	Points to the destination string.
`s2`	Points to the source string.
`n`	Specifies the number of characters write.

Return Value

This function returns the value of `s1`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

strpbrk

Searches a string for a character in a second string.

```
#include <string.h>
char *strpbrk(
   const char *s1,   /* search string */
   const char *s2    /* reference string */
   )
```

Description

This function locates the first occurrence in one string (`s1`) of any character in another string (`s2`).

Arguments

`s1`	Points to the string to be searched.
`s2`	Points to the reference string.

Return Value

The function returns a pointer to the first matching character. If no character from `s2` is found in `s1`, the function returns a null pointer.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strrchr

Searches a string for a character.

```
#include <string.h>
char *strrchr(
   const char *s,    /* search string */
   int c             /* reference character */
   )
```


Description

This function locates the last occurrence of `c` (converted to a `char`) in a string (`s`). The terminating null character is considered part of the string.

Arguments

`s`	Points to the string to be searched.
`c`	Specifies the reference character.

Return Value

This function returns a pointer to the character. If `c` is not found, the function returns a null pointer.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strspn Calculates specified string length.

```
#include <string.h>
size_t strspn(
   const char *s1,   /* candidate string */
   const char *s2    /* reference string */
   )
```

Description

The `strspn()` function computes the length of the maximum initial segment of a string (`s1`) that consists entirely of characters from another string (`s2`).

Arguments

`s1`	Points to the string to be examined.
`s2`	Points to the reference string.

Return Value

This function returns the length of the segment.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

strstr

Searches a string for specified characters in another string.

```
#include <string.h>
char *strstr(
   const char *s1,   /* search string */
   const char *s2    /* reference string */
   )
```


Description

The `strstr()` function locates the first occurrence in a string (`s1`) of the sequence of characters (excluding the null terminator) in another string (`s2`).

Arguments

`s1`	Points to the string to be searched.
`s2`	Points to the reference string.

Return Value

The function returns a pointer to the located string or a null pointer if the string is not found. If `s2` points to a string with zero length, the function returns `s1`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

strtod

Converts a string to a double.

```
#include <stdlib.h>
double strtod(
   const char *nptr,   /* input string */
   char **endptr       /* string conversion terminator */
   )
```

Description

This function converts the initial portion of a string (`nptr`) to a `double` representation. Leading white spaces are ignored. The string can be in scientific exponential form (for example, +123.45e+67, -123.45E+67). The `strtod()` function stops parsing the string when it detects a character that is inconsistent with a `double` data type.

This function sets `errno` if the converted value is out of the supported range for doubles.

Arguments

`nptr`	Points to the string to be converted.
`endptr`	An output parameter. If `nptr` is null, `strtod()` functions the same as `atof()`. If `nptr` is not null, it points to a pointer to the character in `str` that terminated the scan.

Return Value

This function returns either the converted value or 0 if no conversion occurs. No conversion occurs if the first nonwhite space in `str` is neither a digit nor a decimal point. If the correct value is outside the range of representable values, a plus or minus `HUGE_VAL` is returned. If the correct value would cause underflow, 0 is returned. If either of these two errors occurs, `errno` is set.

Notes

The pREPC+ library returns `double` values (including floating point) in the CPU register pair designated by the compiler to receive a return value of type `double` from a function call when a hardware floating point is *not* selected. Additionally, if the FPU bit is set in the processor type entry of the Node Configuration Table, the

pREPC+ library also places the floating point value in the floating point register designated by the compiler to receive a return value of type `double` when a hardware floating point *is* selected.

Callable From

- Task

Error Codes

Refer to Appendix A.

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

`atoi`, *`atol`*, *`atof`*

strtok

Searches a string for tokens.

```
#include <string.h>
char *strtok(
   char *s1,         /* search string */
   const char *s2    /* delimiter(s) */
   )
```

Description

A series of calls to the `strtok()` function breaks a string (`s1`) into a sequence of *tokens*, each of which is delimited by a character in a second string (`s2`). A token is a sequence of one or more characters. The first call to `strtok()` has `s1` as its first argument, and is followed by calls with a null pointer as their first argument.

On the first call to `strtok()`, `s1` is passed as a pointer to the string to be searched, and `s2` is passed as a pointer to the string that contains the delimiters. The first call searches for the first character in `s1` that is not found in the delimiter set. If such a character is found, it is the start of the first token. If the first call fails to find a character that is not a delimiter, there are no tokens, and a null pointer is returned.

The function then continues the search of `s1` for a character that *is* contained in the delimiter set. If no such character is located, the current token extends to the end of `s1`, and subsequent calls to the function return a null pointer. If a delimiter is located, a null character that terminates the current token overwrites the delimiter. The function saves the pointer to the character that follows. This is where the next search for a token starts.

In subsequent calls, the first argument should be a null pointer. The search for the next token begins from the saved pointer and behaves as described in the preceding paragraph.

The search string `s2` can be changed between calls. This allows `strtok()` to continue to parse the string with a different set of delimiters.

Arguments

`s1`	Points to the string to be searched.
`s2`	Points to the string containing token delimiters.

Return Value

The function returns a pointer to the first character of a token. If no token exists, the function returns a null pointer.

Error Codes

None.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

strtok_r

strtok_r

Searches a string for tokens.

```
#include <string.h>
char *strtok_r(
   char *s,          /* search string */
   const char *sep   /* delimiter(s) */
   char **lasts
   )
```

Description

The function `strtok_r` considers the null-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `sep`. The argument `lasts` points to a user-provided pointer, which points to stored information necessary for `strtok_r()` to continue scanning the same string.

On the first call to `strtok_r()`, `s` points to a null-terminated string, and `sep` points to a null-terminated string of separator characters, and the value pointed to by `lasts` is ignored. The function `strtok_r()` returns a pointer to the first character of the first token, writes a null character into `s` immediately following the returned token, and updates the pointer to which `lasts` points.

In subsequent calls, the first argument (`s`) should be a null pointer. The search for the next token begins from the saved pointer (`lasts`) so that subsequent calls will move through the string `s`, returning successive tokens until no tokens remain. When no token remains in `s`, a NULL pointer is returned.

The separator string `sep` may be different from call to call.

Arguments

`s`	Points to the null-terminated string to be searched.
`sep`	Points to a null-terminated string of separator characters.
`lasts`	Points to the token returned.

Return Value

The function returns a pointer to the token. If no more tokens exist, the function returns a NULL pointer.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

strtok

strtol

Converts a string to a long integer.

```
#include <stdlib.h>
long strtol(
   const char *nptr,/* string */
   char **endptr,   /* string conversion terminator */
   int base         /* conversion base */
   )
```

Description

The `strtol()` function converts the initial portion of a string (`nptr`) to `long int` representation, according to the radix specified by `base`. This function ignores leading white spaces, and the string can contain either a + or a -.

Arguments

`nptr`	Points to the string to be converted.
`endptr`	An output parameter. If `endptr` is null, this function is equivalent to the `atol()` function. If `endptr` is not null, it points to a pointer to the character in `str` that terminated the scan.
`base`	Specifies the base of the number system assumed by the function. `base` must be either 0 or within the range 2 through 36. If it is 0, the string itself is used to determine its base. If `nptr` begins with the character 0, base eight is assumed; if `nptr` begins with 0x or 0X, base sixteen is assumed; otherwise base ten is assumed.

Return Value

This function returns the converted value. If the conversion fails, this function returns a 0 and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

atoi, *atof*

strtoul

Converts a string to an unsigned long.

```
#include <stdlib.h>
unsigned long strtoul(
   const char *nptr,/* string */
   char **endptr,   /* string conversion terminator*/
   int base         /* conversion base */
   )
```

Description

This function is equivalent to `strtol()` except that the minus sign (-) is not a valid character to `strtoul()` and the string is converted to an unsigned `long` representation.

Arguments

`nptr`	Points to the string to be converted.
`endptr`	An output parameter. If `endptr` is null, this function is equivalent to the `atol()` function. If `endptr` is not null, it points to a pointer to the character in `nptr` that terminated the scan.
`base`	Specifies the base of the number system assumed by the function. `base` must be either 0 or within the range 2 through 36. If it is 0, the string itself is used to determine its base. If `nptr` begins with the character 0, base eight is assumed; if `nptr` begins with 0x or 0X, base sixteen is assumed; otherwise base ten is assumed.

Return Value

This function returns the converted value. If an error occurs, this function returns a 0, and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = NO

See Also

atol

strxfrm

Transforms a string so that it can be used by the `strcmp()` function.

```
#include <string.h>
size_t strxfrm (
   char *s1,         /* destination string */
   const char *s2    /* source string */
   size_t n          /* destination string length */
   )
```

Description

The `strxfrm()` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the `strcmp()` function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll()` function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The main use for this function is in foreign language environments that do not use the ASCII collating sequence.

Arguments

`s1`	Points to the array where `strxfrm()` places the resulting string.
`s2`	Points to the string to be transformed.
`n`	Specifies the number of characters to be placed in `s1`.

Return Value

The `strxfrm()` function returns the length of the transformed string (not including the terminating null character.) If the value is `n` or more, the contents of the array pointed to by `s1` are indeterminate.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

setlocale, *strcoll*, *strcmp*

tan

Computes the tangent of x (measured in radians)

```
#include <math.h>
double tan (
   double x
   )
```

Description

The `tan` function computes the principal value of the tan of x (measured in radians).

Arguments

x	The angle (measured in radians) whose tangent is to be computed.

Return Value

`tan` returns the tangent of the input number.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

atan, *tanh*

tanh

Computes the hyperbolic tangent of x.

```
#include <math.h>
double tanh (
   double x
   )
```

Description

The tanh function computes the hyperbolic tangent of x.

Arguments

x	The number whose hyperbolic tangent is to be computed.

Return Value

tanh returns the hyperbolic tangent of the input number.

Error Codes

None.

Notes

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

tan, *atan*

time

Obtains the current calendar time.

```
#include <time.h>
time_t time (
   time_t *timer    /* buffer */
   )
```

Description

The `time()` function determines the current calendar time, expressed as the number of seconds since midnight January 1, 1970.

Arguments

`timer` Points to the buffer where `time()` can store the current calendar time.

Return Value

The `time()` function returns the implementation's best approximation of the current calendar time. The value `(time_t) -1` is returned if the calendar time is not available. If `timer` is not a null pointer, the return value is also assigned to the object it points to.

Error Codes

Refer to Appendix A.

Notes

This function invokes the pSOS+ service call `tm_get()` to obtain the current time.

Callable From

- Task
- ISR

Required SC_PREPC Setting

`SC_PREPC = NO`

See Also

tm_get, *mktime*, *localtime*, *localtime_r*

tmpfile

Creates a temporary file.

```
#include <stdarg.h>
#include <stdio.h>
FILE *tmpfile(
   void
   )
```

Description

The `tmpfile()` function creates a temporary file that is automatically removed when it is closed or when the creating task calls `exit()`. Temporary files are opened in mode `wb+`.

The name of the temporary file created is obtained by generating an internal call to the pREPC+ function `tmpname()`.

Return Value

This function returns a pointer to the stream of the file or a null pointer if no file is created. If an error occurs, the function sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

tmpnam

tmpnam

Generates a temporary filename.

```
#include <stdarg.h>
#include <stdio.h>
char *tmpname(
   char *s   /* string root */
   )
```

Description

The `tmpnam()` function generates a string that is intended to be used as a filename. `tmpname()` generates up to 1000 unique names for each task. The pREPC+ library does not maintain a list of `tmpnames` in use. After 1000 names have been generated, the sequence starts over.

A file with a name generated by `tmpnam()` is not necessarily a temporary file. To be treated as a temporary file, it must be created by `tmpfile()`.

Arguments

`s` Points to the string where `tmpname()` stores the filename.

When `tmpname()` is called, `s` should consist of the initial part of a valid pathname. `tmpname()` adds a slash (/), followed by a `T` (or `G` if the creating task is global). The `T` (or `G`) is followed by the lower 16-bits of the caller's task ID, which is followed by a decimal point and a decimal number within the range 0 through 999.

For example, assume that `s` points to the string 0.0; the caller's tid is 00000002; and the caller has previously called `tmpname()` 12 times. The generated name is `0.0/T0002.012`.

Return Value

This function returns a pointer to the generated name.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

tolower

Converts a character to lowercase.

```
#include <ctype.h>
int tolower(
   int c   /* character */
   )
```

Description

This function converts an uppercase letter to lowercase.

Arguments

c	Specifies the character to be converted.

Return Value

This function returns the converted character. If `c` is not an uppercase character, the function returns the character unchanged.

Error Codes

None.

Notes

The `tolower` function is also defined as a macro in the include file `ctype.h`. To use the function instead of the macro, you must add the following statement in your source file anywhere after the `include ctype.h` statement, but before a call to the function:

```
#undef tolower
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

toupper Converts a character to uppercase.

```
#include <ctype.h>
int toupper(
   int c   /* character */
   )
```

Description

This function converts a lowercase letter to uppercase.

Arguments

c	Specifies the character to be converted.

Return Value

This function returns the converted character. If c is not a lowercase character, it is returned unchanged.

Error Codes

None.

Notes

The toupper function is also defined as a macro in the include file ctype.h. To use the function instead of the macro, you must add the following statement in your source file anywhere after the include ctype.h statement, but before a call to the function:

```
#undef toupper
```

Callable From

- Task
- ISR

Required SC_PREPC Setting

```
SC_PREPC = NO
```

ungetc

Ungets a character.

```
#include <stdarg.h>
#include <stdio.h>
int ungetc(
   int c,          /* character */
   FILE *stream    /* stream pointer */
   )
```


Description

The `ungetc()` function pushes a character (`c`), converted to an `unsigned char`, back to the specified stream. The character is returned on the next read operation on the stream. A call to `fseek()`, `fsetpos()`, `rewind()`, or `fflush()` ignores the character.

The stream's position indicator is not changed by this call.

Arguments

`c`	Specifies the character to be pushed.
`stream`	Points to an open pREPC+ stream.

Return Value

This function returns the contents of `c`. If an error occurs, the function returns `EOF` and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

putc, *getc*

vfprintf

Writes formatted output to a stream.

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(
   FILE *stream,          /* stream pointer */
   const char *format,    /* format control */
   va_list arg            /* argument list */
   )
```


Description

The `vfprintf()` function is equivalent to `fprintf()`, with the variable argument list replaced by `arg`, which should have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf function` does not invoke the `va_end` macro.

Arguments

`stream`	Points to an open pREPC+ stream.
`format`	Points to the format control string. For more information, see `fprintf` on page 4-66.
`arg`	A list of arguments to be written according to the specifications of the format control string.

Return Value

This function returns the number of characters written. If a write error occurs, this function returns a negative number and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

fprintf, *vprintf*, *vsprintf*

vprintf

Writes formatted output to stdout.

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(
   const char *format,   /* format control */
   va_list char arg      /* argument list */
   )
```


Description

The vprintf() function is equivalent to printf(), with the variable argument list replaced by arg, which should have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro.

Arguments

format	Points to the format control string. For more information, see fprintf on page 4-66.
arg	A list of arguments to be written according to the specifications of the format control string.

Return Value

This function returns the number of characters written. If a write error occurs, this function returns a negative number and sets errno.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

printf, *vfprintf*, *fprintf*, *vsprintf*

vsprintf

Writes formatted output to a buffer.

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(
   char *s,              /* target buffer */
   const char *format,   /* format control */
   va_list char arg      /* argument list */
   )
```


Description

The `vsprintf()` function is equivalent to `sprintf()`, with the variable argument list replaced by `arg`, which should have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf` function does not invoke the `va_end` macro.

Arguments

`s`	Points to the buffer where output is directed.
`format`	Points to the format control string. For more information, see `fprintf` on page 4-66.
`arg`	A list of arguments to be written according to the specifications of the format control string.

Return Value

This function returns the number of characters written. If a write error occurs, this function returns a negative number and sets `errno`.

Error Codes

Refer to Appendix A.

Notes

Callable From

- Task

Required SC_PREPC Setting

SC_PREPC = YES

See Also

printf, *sprintf*, *fprintf*

wcstombs

Converts a wide character string into a multibyte character string.

```
#include <stdlib.h>
size_t wstombs (
   char *s,                /* result string */
   const wchar_t *pwcs,    /* wide string */
   size_t n                /* size of result string */
   )
```


Description

The `wcstombs()` function converts a sequence of codes that correspond to multibyte characters from the array pointed to by `pwcs` into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters in the array pointed to by `s`, stopping if a multibyte character would exceed the limit of `n` total bytes or if a null character is stored. Each call is converted as if by a call to the `wctomb()` function, except that the shift state of the `wctomb()` function is not affected.

No more than `n` bytes will be modified in the array pointed to by `s`. If copying takes place between objects that overlap, the behavior is undefined.

Arguments

`s`	Points to the array where `wstombs()` stores the converted string.
`pwcs`	Points to the string to be converted.
`n`	Specifies the size of `s`.

Return Value

If a code is encountered that does not correspond to a valid multibyte character, this function returns `(size_t -1.)` Otherwise, it returns the number of bytes modified, not including a terminating null character, if any.

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

wctomb, *mbtowc*, *mbstowcs*

wctomb

Converts a wide character into its multibyte character equivalent.

```
#include <stdlib.h>
int wctomb (
   char *s,          /* result character */
   wchar_t wchar     /* wide character */
   )
```


Description

The wctomb() function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is wchar (including any change in shift state). It stores the multibyte character representation in the array object pointed to by s (if s is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of wchar is zero, the wctomb() function is left in its initial state.

Arguments

s	Points to the array where wctomb() stores the converted character.
wchar	Points to the character to be converted.

Return Value

If s is a null pointer, this function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, this function returns -1 if the value of wchar does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of wchar.

In no case will the value returned be greater than the value of the MB_CUR_MAX macro.

Error Codes

None.

Notes

pREPC+ supports only the “C” locale defined by ANSI. The “C” locale does not support multi-byte encoding; under the “C” locale, the size of a multi-byte character is one byte.

Callable From

- Task
- ISR

Required SC_PREPC Setting

SC_PREPC = NO

See Also

mbtowc, *wcstombs*, *mbstowcs*

5 pLM+ System Calls

This chapter provides detailed information on each system call in the library manager (pLM+) component of pSOSystem. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, a detailed description, its arguments, and its return value. Where applicable, the section also includes the headings "Notes" and "See Also." "Notes" provides important information not specifically related to the call's description, and "See Also" indicates other calls that have related information.

If you need to look up a system call by its functionality, refer to Table 5-1 on page 5-2, which lists the calls alphabetically and provides a brief description of each call.

For more information on error codes, refer to Appendix A, *Error Codes*, which lists the codes numerically and shows which pLM+ System calls are associated with each one.

TABLE 5-1 pLM+ System Calls

Name	Description	Page
sl_acquire	Acquires a shared library for the calling task.	5-3
sl_bindindex	Gets the library's index for the data section variable in the stub file.	5-10
sl_getattr	Obtains the attributes of a registered library.	5-11
sl_getindex	Obtains the index of a registered shared library.	5-13
sl_getsymaddr	Obtains the address of the symbol defined within a registered library,	5-15
sl_register	Adds a shared library to the pLM+ table.	5-17
sl_release	Releases a previously acquired shared library index by the calling task.	5-23
sl_setattr	Sets the writable attributes of a registered library.	5-28
sl_unregister	Unregisters a shared library from the pLM+ table.	5-30
sl_update	Replaces a currently registered library with another library with the same name.	5-34

sl_acquire

Acquires a shared library for the calling task.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_acquire (
   const char *libname,   /* name of the shared library */
   unsigned long scope,   /* scope */
   unsigned long version, /* version of the shared library*/
   const void *libinfo,   /* info for LM_LOADCO callout */
   unsigned long *index   /* index of shared library */
   )
```


Description

The `sl_acquire()` function acquires a shared library (the root shared library) and all of its dependent libraries, recursively until there are no more dependents. The inverse of `sl_acquire()` is `sl_release()`, which releases a shared library and all of its dependents.

Acquired means that the library and all of its dependent libraries are ready to call and cannot be unregistered until they are released by the calling task.

It is only necessary for a task to acquire a shared library once. This can be as either a root shared library or as a dependent shared library. However, a task can acquire a shared library multiple times. To fully release the shared library, it and all shared libraries that depend on it must be released the number of times that each one was acquired as a root shared library.

Acquiring a shared library includes three steps. All steps are performed for the shared library and all of its dependent shared libraries. These steps are named and listed below. The first two steps are done only if not already done. The last step is always done.

- Register
  - Search the table of registered shared libraries by library name. If the shared library is not already registered:
    - Call `LM_LOADCO` to load the shared library. It returns the address of the shared library, its loadhandle, and its unloadmode.
  - Check whether the shared library's version meets the version requirements. For the root shared library, these are specified by the `scope` and `version`

parameters of `sl_acquire()`. For the dependent libraries, these are specified by the shared library dependency within the parent shared library's library header. You can receive the error `LME_MAJORVER` or `LME_MINORVER` if the version requirements are not met.

- Search the table of registered shared libraries by library key. Abort with error `LME_KEYCOLL` if a registered shared library has the same library key as the newly loaded shared library.

  NOTE: The shared library key is used to optimize the calling of a shared library via a stub file. Every registered shared library has a unique key. The key is assigned by the `shlib` host tool when the shared library is built. The key is a hash of the shared library name, or can be overridden in the shared library's library definition file. The resulting key is recorded both in the shared library and in the stub file used to call the shared library.

- Add the shared library to pLM+'s table of loaded libraries. This assigns the library an index. You will receive the error `LME_MAXREG` if the table is full.

- Call the shared library's entry function, if any, with `SL_REGEV` to do global initialization.

  This step is done the nearly same way as `sl_register()`. The dependent shared libraries are done exactly the same. The root shared library is done differently due to the different parameters of `sl_acquire()` and `sl_register()`.

■ Attach - If the library is not already acquired by the calling task attach the shared library as follows:

  - Call the shared library's entry function, if any, with `SL_ATTACHEV` to do per-task initialization.

  - Increment the shared library's count of attached tasks. This is the `proc_count` member of `attr` returned by `sl_getattr()`.

■ Acquire - This step is always done.

  - Increment the shared library's count of acquires by the calling task. This is the `acq_count` member of `attr` returned by `sl_getattr()`.

The `version` and `scope` parameters of `sl_acquire()` specify the caller's version requirements for the root shared library. In the same way, a shared library's dependencies within its header specify the dependent shared libraries' name and the ver-

sion requirements upon them in terms of version and scope. A shared library version number is composed of a 16-bit major version, and a 16-bit minor version. The scope has three possible values: SL_ANY, SL_COMPATIBLE, and SL_EXACT. SL_ANY matches any library version. SL_COMPATIBLE matches a library version with the same major version as the version parameter, and a minor version equal to or higher than the minor version of the version parameter. SL_EXACT matches only the exact same version as the version parameter.

The shared libraries that were not already registered are loaded as follows. For each shared library to load, LM_LOADCO is called in the context of the calling task. For the root shared library, the LM_LOADCO parameter values libname, scope, version, and libinfo are the corresponding parameters of sl_acquire(). The libinfo parameter is a way to pass additional information to LM_LOADCO. For example, it could be the directory from which to load shared libraries, or the IP address of the server from which to load shared libraries. For the dependent shared libraries, the LM_LOADCO parameter values are derived from the dependencies recorded within the library headers of the parent libraries. These dependencies came from the USE statements of the parent libraries' library definition files. The libname parameter is simply taken from the dependency. The scope and version parameters are chosen to fulfill the scope and version requirements of all dependencies upon the same named library read at the time the dependent library is loaded. This does not take into account any additional dependencies found as later dependent libraries are loaded. The libinfo parameter is chosen from one of the dependencies on the named library. It is not defined which one. The libinfo is less useful for the dependent libraries since it is set when the parent shared libraries are built, not at execution time as for the root shared library. Therefore, it might be better to use the root shared library, libinfo, for all libraries loaded. None of the library dependencies would have a libinfo parameter. Instead, LM_LOADCO would save the libinfo parameter, if not NULL, in a local variable and use it to load both the root and dependent libraries.

LM_LOADCO returns a 0 if successful, or an error code if it fails. If it fails, pLM+ translates that to error LME_LOADCO. There is not a way to determine the actual error code returned by LM_LOADCO. If that is needed, LM_LOADCO should be written to record the error code somewhere as well as return it to pLM+.

LM_LOADCO, if successful, returns in sl_attrib the address of the shared library, its unloadmode, and its loadhandle. The loadhandle is a way LM_LOADCO can pass additional information to LM_UNLOADCO. The loadhandle is saved and passed to LM_UNLOADCO if the shared library is unloaded. For example, if LM_LOADCO uses the pSOSystem target loader, the loadhandle could be the OF_INFO* needed to unload the shared library.

The entry function, if any, of each library registered or attached is called in the context of the calling task. The index parameter is set to the shared library's index. The event parameter is set to SL_REGEV or SL_ATTACHEV for register or attach, respectively. If a shared library is both registered and attached, the entry function will be called twice, first with event SL_REGEV and later with event SL_ATTACHEV. The index parameter of the entry function may be stored in the library for later use, but cannot be used in a pLM+ system call until the registration function completes successfully. The entry function returns 0 if it succeeds or anything else if it fails. If the entry function fails, sl_acquire() translates that to error LME_REGEV or LME_ATTACHEV for register or attach, respectively. There is not a way to determine the actual error code returned by the entry function. If that is needed the entry function should be written to record the error code somewhere as well as return it to pLM+.

The registration system calls, that is, sl_acquire(), sl_register(), and sl_bindindex(), are atomic with one exception concerning sl_acquire(). While a registration system call is executing, indices of any newly registered shared libraries, if any, are invalid and cannot be used. All of these indices become valid together at the end of the registration system call. If an error occurs, all completed and partially completed steps are undone for the shared library and all its dependents. The process of undoing these steps is very similar to the steps performed by the sl_release() function, for sl_acquire(), or sl_unregister(), for the others.

The one exception to atomicity of sl_acquire() is the counts returned by sl_getattr(). If a sl_getattr() call is made for a shared library that is being acquired by a concurrent call to sl_acquire(), and that shared library was already registered before the sl_acquire() call, the counts returned by the sl_getattr() call can be the values incremented by the sl_acquire() call even if the sl_acquire() call eventually fails due to an error and undoes all its work. The overhead of atomicity for these counts is not worth it.

A shared library entry function or a pLM+ callout may not call a pLM+ system call that will register or unregister a shared library if they themselves are called from a pLM+ system call that is registering or unregistering a shared library. Attempting to do so causes error LME_REENTER. This always applies to pLM+ callouts, and to entry functions called with events SL_REGEV or SL_UNREGEV, since they are always called from pLM+ system calls that are registering or unregistering a shared library. It sometimes applies to entry functions called with events SL_ATTACHEV or SL_DETACHEV since they are sometimes but not always called from pLM+ system calls that are registering or unregistering a shared library. The disallowed calls are: sl_register(), sl_unregister(), sl_acquire() or sl_bindindex() if the specified library is not already registered, and sl_release() if the specified library will be unregistered; that is, it will be fully released by all tasks and does not have

unloadmode SL_MANUAL. Together these disallow calling other shared libraries unless they are registered already.

Error LME_REENTER can be prevented by the following restrictions. LM_LOADCO, LM_UNLOADCO, and shared library entry functions called with events SL_REGEV or SL_UNREGEV should not call other shared libraries, sl_register(), sl_acquire(), sl_unregister(), or sl_release(). Shared library entry functions called with SL_ATTACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire(). Shared library functions called with SL_DETACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire() and all of the shared libraries have unloadmode SL_MANUAL so they are not unregistered by sl_release(). If a shared library entry function called with SL_REGEV or SL_ATTACHEV needs to call other shared libraries and the above restrictions cannot be met, the entry function can be written to set a flag in the shared library's data area, indicating that a register or attach needs to be done. Then, all of the functions exported by the shared library would check these flags and do the deferred register or attach initialization before their normal processing.

All the pLM+ functions are designed for a multitasking environment. Concurrent pLM+ system calls wait if necessary to maintain consistency or provide a sequential ordering of pLM+ system calls. For example, if the index of a newly registered shared library is used in a system call that has a library index parameter before the index is valid, the system call will wait for the registration system call to finish.

The sl_register() function is an alternative to sl_acquire(). It does only the register step. Like sl_acquire(), it registers the shared library and all of its dependents. If a shared library or one of its dependents has an entry function that does per-task initialization, sl_acquire() must be used. The shared library will not function correctly if only sl_register() is used.

Arguments

libname	Specifies the name of the shared library.
scope	The scope parameter can be set to any one of the following:
	SL_EXACT — Checks for the exact match of the major and minor version numbers.
	SL_COMP — Checks for exact major versions and equal or higher minor versions (revisions) of the registered library.
	SL_ANY — Ignores the version number and any library with the same name is accepted.
version	Specifies the version number of the shared library.
libinfo	Provides additional information for LM_LOADCO if it is called to load the root-shared library.
index	Returns the index of the root-shared library.

Return Value

This system call returns 0 on success, or an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The returned library's version may be different than the requested version in the case of SL_COMP or SL_ANY scopes. The version number returned can be obtained by calling sl_getattr() with the index returned by sl_acquire().
2. Many errors can occur during the acquisition of dependent libraries. When multiple errors occur, sl_acquire() always returns the error that is encountered first.
3. pLM+ only registers one version of each dependent library. The dependent libraries can have multiple dependencies on the same named library. If these have conflicting version requirements, sl_acquire() will return an error. If there is a dependency on a shared library that is already registered and the

dependency conflicts with the version of that shared library, `sl_acquire()` will return an error.

4. The ability to acquire a shared library multiple times allows many different programming styles to call a shared library. All needed shared libraries could be acquired at the start of the `ROOT` task or portion of an application that uses them, and all released at the end of the `ROOT` task or application portion. Alternately, a function that calls a shared library could acquire the shared library before calling it, and release it afterwards. Many other styles are also possible. For efficiency, if possible, acquire and release a shared library only once, or at least as few times as possible.

5. `LM_LOADCO` does not necessarily load anything. If the shared library is part of the pSOSystem image, `LM_LOADCO` need only look up the address of the shared library in a table, and return that value. On the other hand, if the shared library is not part of the pSOSystem image `LM_LOADCO` does actually load the shared library. The pSOSystem loader or any other loader can be used. (See "Loader" in Chapter 1, "System Services" in the *pSOSystem Programmer's Reference* manual.)

Multiprocessor Considerations

None.

See Also

sl_release, *sl_register*, *sl_getsymaddr*

sl_bindindex

Gets the library's index for the data section variable in the stub file.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_bindindex(
   const char *libname,     /* shared library name */
   unsigned long version,   /* shared library version */
   void *libinfo,           /* information for LM_BINDCO */
   unsigned long *index     /* shared library index */
   )
```

Description

The `sl_bindindex()` function is only used in stub files. It is not callable from C. It performs the same operation as `sl_register()` with scope `SL_COMP`.

Arguments

`libname`	Specifies the shared library name.
`version`	Specifies the version number of the library.
`libinfo`	Specifies the information on the registered library.
`index`	Returns the index of the shared library.

Return Value

This function returns a 0 on success and an error code on failure.

Error Codes

Refer to Appendix A.

See Also

sl_register

sl_getattr

Obtains the attributes of a registered library.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_getattr(
   unsigned long index,   /* index of shared library */
   sl_attrib *attr        /* library attributes */
   )
```

Description

The function `sl_getattr()` verifies that `index` represents a valid registered library. If so, it returns the attributes of the shared library corresponding to `index` by storing them into the structure pointed to by `attr`.

The structure `sl_attrib` has the following fields:

```
const char *name;
ULONG version;
const sl_header *libaddr;   /* address of library header */
ULONG proc_count;           /* number of attached tasks */
ULONG acq_count;            /* number of times acquired by caller */
ULONG unloadmode;          /* manual/autounreg/autounreg & unload */
const void *loadhandle;     /* load handle */
```

You can obtain the attributes of all the currently registered libraries by looping, calling `sl_getattr()` with the loop index, starting at zero and incrementing each time until `LME_BIGINDEX` is returned. If any of the indices within this range are unused, `sl_getattr()` will return `LME_INVINDEX` and those indices can be skipped.

Arguments

`index`	Points to the variable which stores the index of the shared library.
`attr`	Returns library attributes.

Return Value

This function returns a 0 on success and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

See Also

sl_setattr, *sl_getindex*, *sl_getsymaddr*

sl_getindex

Obtains the index of a registered shared library.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_getindex(
   const char *libname,   /* name of the shared library */
   unsigned long scope,   /* see below */
   unsigned long version, /* version of the shared library */
   unsigned long *index   /* index of the shared library */
)
```


Description

The `sl_getindex()` function searches the pLM+'s table for a registered library having the same name as specified by `libname` and a suitable version number specified by `version` depending on `scope`.

Arguments

`libname` Pointer to the name of the library

`scope` The `scope` parameter can be set to any one of the following:

- `SL_EXACT` Checks for the exact match of the major and minor version numbers.
- `SL_COMP` Checks for the exact match of the major versions and equal or higher minor versions (revisions) of the registered library.
- `SL_ANY` Ignores the version number and any library with the same name is accepted. If suitable library is found, it returns its index by storing it into the variable pointed to by index. This index may then be used to reference the library in other pLM+ calls.

`version` Specifies the version of the shared library.

`index` Pointer to the index of the shared library.

Return Value

On success this function returns 0, and it returns an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. The returned library's version may be different than the requested version in the case of `SL_COMP` or `SL_ANY` scopes and it can be obtained by calling `sl_getattr()` with the index returned by `sl_getindex()`.
2. `sl_getindex()` does not acquire or register the shared library. You must use `sl_acquire()` or `sl_register()`, respectively, for those purposes.
3. The `sl_getindex()` function does not check whether the calling task has acquired the shared library or not. If the shared library is acquired by the task using `sl_acquire()` before `sl_getindex()` is called, then you can get the index of the shared library by calling `sl_getindex()` by passing it the same name, version and scope parameters as used in `sl_acquire()`. This is possible since only one version of a library is allowed to be registered with pLM+ at any time.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

`sl_register`, *`sl_acquire`*, *`sl_getsymaddr`*

sl_getsymaddr

Obtains the address of the symbol defined within a registered library.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_getsymaddr(
   unsigned long index,    /* index of shared library */
   const char *symname,    /* name of the symbol */
   void **symaddr          /* address of the symbol */
   )
```


Description

The `sl_getsymaddr()` function first verifies that `index` represents a valid registered library. If so, it searches the symbol table within the header of the shared library corresponding to `index` for a symbol named `symname`. If the symbol is found, the address of the symbol is returned by storing it into the variable pointed to by `symaddr`.

The `sl_getsymaddr()` function is intended for use primarily when calling a shared library function explicitly. `sl_getsymaddr()` does not check whether the calling task has acquired the shared library corresponding to index or not.

Arguments

`index`	Specifies the index of shared library.
`symname`	Points to the name of the symbol.
`symaddr`	Pointer to the variable which stores the address of the symbol returned by this function.

Return Value

This function returns a 0 on success and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

See Also

sl_getindex, *sl_getattr*

sl_register

Adds a shared library to the pLM+ table.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_register(
   const sl_header *libaddr,   /* shared library header*/
   const sl_attrib *attr,      /* library attributes */
   unsigned long *index        /* shared library index */
   )
```

Description

The sl_register() function registers a shared library (the root shared library) and all of its dependent shared libraries, recursively until there are no more dependents. The inverse of sl_register() is sl_unregister() which unregisters a shared library and all of its dependents.

Unlike sl_acquire(), the shared libraries are neither attached nor acquired. Any per-task shared library initialization is not done. The shared library is not protected from being unregistered. If either or both of these two additional features are needed, call sl_acquire(), either instead of sl_register() or after sl_register().

A shared library can only be registered once. If the shared library is already registered, error LME_DUPLIB is returned.

Registering a shared library includes only one step: register. This is performed for the shared library and all of its dependent shared libraries.

- Register
  - For the root shared library
    - Search the table of registered shared libraries by library name. If the root shared library is already registered, abort with error LME_DUPLIB.
- For the dependent shared libraries
  - Search the table of registered shared libraries by library name. If the shared library is not already registered:
    - Call LM_LOADCO to load the shared library. It returns the address of the shared library, its loadhandle, and its unloadmode.

- Check whether the shared library's version meets the version requirements specified by the shared library dependency within the parent shared library's library header. You can receive the error LME_MAJORVER or LME_MINORVER if the version requirements are not met.

■ For both the root and the dependent shared libraries

- Search the table of registered shared libraries by library key. Abort with error LME_KEYCOLL if a registered shared library has the same library key as the newly loaded shared library.

  NOTE: The shared library key is used to optimize the calling of a shared library via a stub file. Every registered shared library has a unique key. The key is assigned by the shlib host tool when the shared library is built. The key is a hash of the shared library name, or can be overridden in the shared library's library definition file. The resulting key is recorded both in the shared library and in the stub file used to call the shared library.

- Add the shared library to pLM+'s table of loaded libraries. This assigns the library an index. You will receive the error LME_MAXREG if the table is full.

- Call the shared library's entry function, if any, with SL_REGEV to do global initialization.

This step is done nearly the same way as sl_acquire(). The dependent shared libraries are done exactly the same. The root shared library is done differently due to the different parameters of sl_acquire() and sl_register().

A shared library's dependencies within its header specify the dependent shared libraries' name and the version requirements upon them in terms of version and scope. A shared library version number is composed of a 16-bit major version, and a 16-bit minor version. Scope has three possible values: SL_ANY, SL_COMPATIBLE, and SL_EXACT. SL_ANY matches any library version. SL_COMPATIBLE matches a library version with the same major version as the version parameter, and a minor version equal to or higher than the minor version of the version parameter. SL_EXACT matches only the exact same version as the version parameter.

The dependent shared libraries that were not already registered are loaded as follows. For each shared library to load, LM_LOADCO is called in the context of the calling task. The LM_LOADCO parameter values are derived from the dependencies recorded within the library headers of the parent libraries. These dependencies came from the USE statements of the parent libraries' library definition files. The libname parameter is simply taken from the dependency. The scope and version

parameters are chosen to fulfill the scope and version requirements of all dependencies upon the same named library read at the time the dependent library is loaded. This does not take into account any additional dependencies found as later dependent libraries are loaded. The libinfo parameter is chosen from one of the dependencies on the named library. It is not defined which one. The libinfo parameter is a way to pass additional information to LM_LOADCO. For example, it could be the directory from which to load shared libraries, or the IP address of the server from which to load shared libraries. However, libinfo is more useful for the root shared library in sl_acquire() than the dependent libraries in either sl_register() or sl_acquire(). It is set at execution time for the root shared library of sl_acquire(). For dependent libraries, it is set when the parent shared libraries are built, not at execution time.

LM_LOADCO returns a 0 if successful, or an error code if it fails. If it fails, pLM+ translates that to error LME_LOADCO. There is not a way to determine the actual error code returned by LM_LOADCO. If that is needed, LM_LOADCO should be written to record the error code somewhere as well as return it to pLM+.

LM_LOADCO, if successful, returns in sl_attrib the address of the shared library, its unloadmode, and its loadhandle. The loadhandle is a way LM_LOADCO can pass additional information to LM_UNLOADCO. The loadhandle is saved and passed to the LM_UNLOADCO if the shared library is unloaded. For example, if LM_LOADCO uses the pSOSystem target loader, the loadhandle could be the OF_INFO* needed to unload the shared library.

The LM_LOADCO is never called for the root shared library. The root shared library's address is parameter libaddr of sl_register(). If the sl_register() parameter attr is not NULL, the root shared library's loadhandle and unloadmode are set to the corresponding fields of attr. Otherwise, the root shared library's loadhandle is set to NULL, and unloadmode to SL_MANUAL.

The entry function, if any, of each library registered is called in the context of the calling task. The index parameter is set to the shared library's index. The event parameter is set to SL_REGEV. The index parameter of the entry function may be stored in the library for later use, but it is not valid until the registration function completes successfully. The entry function returns 0 if it succeeds or anything else if it fails. If the entry function fails, sl_register() translates that to error LME_REGEV. There is not a way to determine the actual error code returned by the entry function. If that is needed, the entry function should be written to record the error code somewhere as well as return it to pLM+.

The registration system calls, that is, sl_acquire(), sl_register(), and sl_bindindex(), are atomic with one exception concerning sl_acquire(). While

a registration system call is executing, indices of any newly registered shared libraries, if any, are invalid and cannot be used. All these indices become valid together at the end of the registration system call. If an error occurs, all completed and partially completed steps are undone for the shared library and all of its dependents. The process of undoing these steps is very similar to the steps performed by the `sl_release()` function, for `sl_acquire()`, or `sl_unregister()`, for the others.

A shared library entry function or a pLM+ callout may not call a pLM+ system call that will register or unregister a shared library if they themselves are called from a pLM+ system call that is registering or unregistering a shared library. Attempting to do so causes error `LME_REENTER`. This always applies to pLM+ callouts, and to entry functions called with events `SL_REGEV` or `SL_UNREGEV`, since they are always called from pLM+ system calls that are registering or unregistering a shared library. It sometimes applies to entry functions called with events `SL_ATTACHEV` or `SL_DETACHEV` since they are sometimes but not always called from pLM+ system calls that are registering or unregistering a shared library. The disallowed calls are: `sl_register()`, `sl_unregister()`, `sl_acquire()` or `sl_bindindex()` if the specified library is not already registered, and `sl_release()` if the specified library will be unregistered; that is, it will be fully released by all tasks and does not have unloadmode `SL_MANUAL`. Together these disallow calling other shared libraries unless they are registered already.

Error `LME_REENTER` can be prevented by the following restrictions. `LM_LOADCO`, `LM_UNLOADCO`, and shared library entry functions called with events `SL_REGEV` or `SL_UNREGEV` should not call other shared libraries, `sl_register()`, `sl_acquire()`, `sl_unregister()`, or `sl_release()`. Shared library entry functions called with `SL_ATTACHEV` should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by `sl_register()` or `sl_bindindex()` before any of them are acquired by `sl_acquire()`. Shared library functions called with `SL_DETACHEV` should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by `sl_register()` or `sl_bindindex()` before any of them are acquired by `sl_acquire()` and all of the shared libraries have unloadmode `SL_MANUAL` so they are not unregistered by `sl_release()`. If a shared library entry function called with `SL_REGEV` or `SL_ATTACHEV` needs to call other shared libraries and the above restrictions cannot be met, the entry function can be written to set a flag in the shared library's data area, indicating that a register or attach needs to be done. Then, all of the functions exported by the shared library would check these flags and do the deferred register or attach initialization before their normal processing.

All the pLM+ functions are designed for a multitasking environment. Concurrent pLM+ system calls wait, if necessary, to maintain consistency or provide a sequen-

tial ordering of pLM+ system calls. For example, if the index of a newly registered shared library is used in a system call that has a library index parameter before the index is valid, the system call will wait for the registration system call to finish.

Arguments

`libaddr`	Points to the shared library header.
`attr`	Points to the shared library attributes or NULL. If not NULL, only the load handle and unloadmode fields are used.
`index`	Points to the index of the shared library.

Return Value

This function returns a 0 on success, and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Many errors can occur during the registration of dependent libraries. When multiple errors occur, `sl_register()` always returns the error that is encountered first.
2. pLM+ only registers one version of each dependent library. The dependent libraries can have multiple dependencies on the same named library. If these have conflicting version requirements, `sl_register()` will return an error. If there is a dependency on a shared library that is already registered, and the dependency conflicts with the version of that shared library, `sl_register()` will return an error.
3. `LM_LOADCO` does not necessarily load anything. If the shared library is part of the pSOSystem image, `LM_LOADCO` need only look up the address of the shared library in a table, and return that value. On the other hand, if the shared library is not part of the pSOSystem image, `LM_LOADCO` does actually load the shared library. The pSOSystem loader or any other loader can be used. (See “Loader” in Chapter 1, “System Services” in the *pSOSystem Programmer’s Reference* manual.)

4. `sl_register()`, but not `sl_acquire()` or `sl_bindindex()`, can be used without any `LM_LOADCO` at all if the shared libraries do not have any cyclic dependencies. Register the shared libraries in an order such that all dependent shared libraries are registered before any shared library that depends upon them. Then, `LM_LOADCO` will never be called, and hence need not exist. For example, if shared library A depends on B and C, B does not depend on any other, C depends on D, and D does not depend on any other, registration in the order D, C, B, A never calls `LM_LOADCO`. If the additional features of `sl_acquire()` are needed, acquire A after it and all of its dependent shared libraries have been registered. Again, `LM_LOADCO` is not called since all of the shared libraries are already registered.

Multiprocessor Considerations

None.

See Also

`sl_unregister`, *`sl_acquire`*, *`sl_getsymaddr`*

sl_release

Releases a previously acquired shared library index by the calling task.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_release(
   unsigned long index   /* index of shared library */
   )
```

Description

The `sl_release()` function releases a shared library (the root shared library) and all of its dependent libraries, recursively until there are no more dependents. The inverse of `sl_release()` is `sl_acquire()`, which acquires a shared library and all of its dependents.

Because a task can acquire a shared library multiple times, to fully release the shared library, it and all shared libraries that depend on it must be released the number of times that each one was acquired as a root shared library.

Releasing a shared library includes three steps. All steps are performed for the shared library and all of its dependent shared libraries. These steps are named and listed below. The first step is always done. The last two steps are done if the shared library is fully released.

- Release - This step is always done.
  - The calling task releases the library. Decrement the shared library's count of acquires by the calling task. This is the `acq_count` member of `attr` returned by `sl_getattr()`.
- Detach - If the library has been fully released by the calling task, detach the shared library as follows:
  - Decrement the shared library's count of attached tasks. This is the `proc_count` member of `attr` returned by `sl_getattr()`.
  - Call the shared library's entry function, if any, with `SL_DETACHEV` to do per-task cleanup.
- Unregister if the library has been fully released by all tasks and the library's unloadmode is `SL_AUTOREG` or `SL_AUTOUNLOAD`; that is, not `SL_MANUAL`:
  - Remove the shared library from pLM+'s table of loaded libraries.

- Call the shared library's entry function, if any, with SL_UNREGEV to do global cleanup.
  - If the shared library's unloadmode is SL_AUTOUNLOAD, call LM_UNLOADCO to unload the shared library.

    This step is done the same way as sl_unregister(). The only difference is sl_release() unregisters a shared library only if enabled by its unloadmode. sl_unregister() unregisters the root shared library regardless of its unloadmode, and its dependents only if enabled by their unloadmodes.

pLM+ guarantees that whenever a shared library is registered, all of its dependents, recursively, are also registered. Therefore, a shared library cannot be unregistered if another registered shared library depends on it, and that other shared library is not being unregistered at the same time. If this happens, this is error LME_LIBINUSE. For example, shared library A depends on B and C, B does not depend on any other, C depends on D, D does not depend on any other, and all are registered. Attempting to unregister any of these libraries, except A, with either sl_unregister() or sl_release() causes error LME_LIBINUSE. An sl_release() call that does not unregister, either because the library is not fully released by all tasks, or the library has unloadmode SL_MANUAL, will not give this error.

The shared libraries that have unloadmode SL_AUTOUNLOAD are unloaded by LM_UNLOADCO. LM_UNLOADCO is called with the loadhandle of the shared library. LM_UNLOADCO returns a 0 if successful, or an error code if it fails. If it fails, pLM+ translates that to error LME_UNLOADCO. There is not a way to determine the actual error code returned by LM_UNLOADCO. If that is needed, LM_UNLOADCO should be written to record the error code somewhere as well as return it to pLM+.

The entry function, if any, of each library detached or unregistered is called in the context of the calling task. The index parameter is set to the shared library's index. The event parameter is set to SL_DETACHEV or SL_UNREGEV for detach or unregister, respectively. If a shared library is both detached and unregistered, the entry function will be called twice, first with event SL_DETACHEV and later with event SL_UNREGEV. The index parameter of the entry function cannot be used within a pLM+ system call. The entry function returns 0 if it succeeds or anything else if it fails. If the entry function fails, sl_release() translates that to error LME_DETACHEV or LME_UNREGEV for detach or unregister, respectively. There is not a way to determine the actual error code returned by the entry function. If that is needed, the entry function should be written to record the error code somewhere as well as return it to pLM+. Even if an error occurs, the sl_unregister() and sl_release() calls always complete and do not abort.

A shared library entry function or a pLM+ callout may not call a pLM+ system call that will register or unregister a shared library if they themselves are called from a pLM+ system call that is registering or unregistering a shared library. Attempting to do so causes error LME_REENTER. This always applies to pLM+ callouts, and to entry functions called with events SL_REGEV or SL_UNREGEV, since they are always called from pLM+ system calls that are registering or unregistering a shared library. It sometimes applies to entry functions called with events SL_ATTACHEV or SL_DETACHEV since they are sometimes but not always called from pLM+ system calls that are registering or unregistering a shared library. The disallowed calls are: sl_register(), sl_unregister(), sl_acquire() or sl_bindindex() if the specified library is not already registered, and sl_release() if the specified library will be unregistered; that is, it will be fully released by all tasks and does not have unloadmode SL_MANUAL. Together these disallow calling other shared libraries unless they are registered already.

Error LME_REENTER can be prevented by the following restrictions. LM_LOADCO, LM_UNLOADCO, and shared library entry functions called with events SL_REGEV or SL_UNREGEV should not call other shared libraries, sl_register(), sl_acquire(), sl_unregister(), or sl_release(). Shared library entry functions called with SL_ATTACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire(). Shared library functions called with SL_DETACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire() and all of the shared libraries have unloadmode SL_MANUAL so they are not unregistered by sl_release(). If a shared library entry function called with SL_REGEV or SL_ATTACHEV needs to call other shared libraries and the above restrictions cannot be met, the entry function can be written to set a flag in the shared library's data area, indicating that a register or attach needs to be done. Then, all of the functions exported by the shared library would check these flags and do the deferred register or attach initialization before their normal processing.

All the pLM+ functions are designed for a multitasking environment. Concurrent pLM+ system calls wait if necessary to maintain consistency or provide a sequential ordering of pLM+ system calls.

Arguments

`index` Points to the index of the shared library.

Return Value

This function returns 0 on success and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Since the same index is returned by `sl_acquire()` on all invocations on the same library, any one index returned by `sl_acquire()` can be used in `sl_release()` to release multiple acquisitions. Also, it is not possible to call `sl_release()` on any index of a library that is not acquired as the root library in a `sl_acquire()` call. This protection prevents accidental removal of a dependent library when some other parent libraries are still using it.

2. Many errors can occur during the release of dependent libraries such as entry function errors, unload callout errors, etc. Also, more than one error condition can occur during one `sl_release()` call, as `sl_release()` ignores the errors and continues to release all the acquired unique dependent libraries of the root library corresponding to index, to avoid orphan libraries. `sl_release()` returns the error that is encountered first.

3. If index value is -1, then `sl_release()` will release all of the pLM+ resources related to the calling task. Also, it forcibly detaches the calling task from all of its acquired libraries even if their counts of the number of times that the task has acquired them are greater than 1 when the call is made. It will call all the associated library entry functions in the context of the calling task, with SL_DETACHEV as the event parameter. This is useful for returning all the pLM+ resources before the task deletes itself.

4. It is not possible to delete a task that still has some acquired libraries without the task detaching from them. When a task is restarted all of its acquired libraries are not detached and remain acquired by the task.

5. For efficiency, if possible, acquire and release a shared library only once, or at least as few times as possible. See `sl_acquire()` for more information.

6. LM_UNLOADCO does not necessarily unload anything. If the shared library is part of the pSOSystem image, LM_UNLOADCO need only look up the address of the shared library in a table, and remove that value from the table. On the other hand, if the shared library is not part of the pSOSystem image, LM_UNLOADCO does actually unload the shared library. (See “Loader” in Chapter 1, “System Services” in the *pSOSystem Programmer’s Reference* manual.)
7. sl_unregister() can be used without any LM_UNLOADCO at all if none of the shared libraries has unloadmode SL_AUTOUNLOAD.

Multiprocessor Considerations

None.

Callable From

- Task

See Also

sl_acquire

sl_setattr

Sets the writable attributes of a registered library.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_setattr(
   unsigned long index,      /* index of shared library */
   const sl_attrib *attrib   /* new library attributes */
   )
```

Description

The sl_setattr() function first verifies that `index` represents a valid registered library. If so, it sets the writable attributes of the shared library corresponding to `index` by reading them from the structure pointed to by `attr`.

The structure `sl_attrib` has the following fields:

```
const char *name;
unsigned long version;
const sl_header *libaddr;  /* address of library header */
unsigned long proc_count;  /* number of attached tasks */
unsigned long acq_count;   /* number of times acquired by caller */
unsigned long unloadmode;  /* manual/autounreg/autounreg & unload */
const void *loadhandle;    /* load handle */
```

Only the unloadmode and loadhandle fields can be set. Other fields are ignored by `sl_setattr()`. unloadmode can be one of the following:

- `SL_MANUAL` - If this mode is set, no automatic actions will be performed and the library will not be unregistered or unloaded when the last task detaches from it.
- `SL_AUTOUNREG` - If this mode is set, the library will be automatically unregistered when the last task detaches from the library (see *sl_release*).
- `SL_AUTOUNLOAD` - If this mode is set, the library will be automatically unregistered when the last task detaches from the library. Also, `LM_UNLOADCO` will be called with the stored library's loadhandle as its parameter to unload the library.

Arguments

`index`	Specifies the index of the shared library.
`attr`	Points to the new library attributes.

Return Value

This function returns a 0 on success, and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

See Also

sl_getattr, *sl_getindex*

sl_unregister

Unregisters a shared library from the pLM+ table.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_unregister(
   unsigned long index   /* index of shared library */
   )
```

Description

The `sl_unregister()` function unregisters a shared library (the root shared library) and all its dependent shared libraries. The inverse of `sl_unregister()` is `sl_register()`, which registers a shared library and all of its dependents, recursively until there are no more dependents.

Unlike `sl_release()`, the shared libraries are not released. To release and unregister the library, call `sl_release()` instead of `sl_unregister()`.

Unregistering a shared library includes only one step: unregister. This is performed for the shared library and all of its dependent shared libraries.

- Unregister - This step is always done for the root shared library. It is done for the dependent libraries only if their unloadmode is `SL_AUTOUNREG` or `SL_AUTOUNLOAD`.
  - Search the table of registered shared libraries by library name. If the root shared library is not registered, abort with error `LME_NOTFOUND`.
  - Remove the shared library from pLM+'s table of loaded libraries.
  - Call the shared library's entry function, if any, with `SL_UNREGEV` to do global cleanup.
  - If the shared library's unloadmode is `SL_AUTOUNLOAD`, call `LM_UNLOADCO` to unload the shared library.

    This step is done the same way as `sl_unrelease()`. The only difference is `sl_release()` unregisters a shared library only if enabled by its unloadmode. `sl_unregister()` unregisters the root shared library regardless of its unloadmode, and its dependents only if enabled by their unloadmodes.

pLM+ guarantees that whenever a shared library is registered, all of its dependents, recursively, are also registered. Therefore, a shared library cannot be unregistered if another registered shared library depends on it, and that other shared library is not

being unregistered at the same time. If this happens, this is error LME_LIBINUSE. For example, shared library A depends on B and C, B does not depend on any other, C depends on D, D does not depend on any other, and all are registered. Attempting to unregister any of these libraries, except A, with either sl_unregister() or sl_release() causes error LME_LIBINUSE. An sl_release() call that does not unregister, either because the library is not fully released by all tasks, or the library has unloadmode SL_MANUAL, will not give this error.

The shared libraries that have unloadmode SL_AUTOUNLOAD are unloaded by LM_UNLOADCO. LM_LOADCO is called with the loadhandle of the shared library. LM_UNLOADCO returns a 0 if successful, or an error code if it fails. If it fails, pLM+ translates that to error LME_UNLOADCO. There is not a way to determine the actual error code returned by LM_UNLOADCO. If that is needed, LM_UNLOADCO should be written to record the error code somewhere as well as return it to pLM+. Even if an error occurs, the sl_unregister() and sl_release() calls always complete and do not abort.

A shared library entry function or a pLM+ callout may not call a pLM+ system call that will register or unregister a shared library if they themselves are called from a pLM+ system call that is registering or unregistering a shared library. Attempting to do so causes error LME_REENTER. This always applies to pLM+ callouts, and to entry functions called with events SL_REGEV or SL_UNREGEV, since they are always called from pLM+ system calls that are registering or unregistering a shared library. It sometimes applies to entry functions called with events SL_ATTACHEV or SL_DETACHEV since they are sometimes but not always called from pLM+ system calls that are registering or unregistering a shared library. The disallowed calls are: sl_register(), sl_unregister(), sl_acquire() or sl_bindindex() if the specified library is not already registered, and sl_release() if the specified library will be unregistered; that is, it will be fully released by all tasks and does not have unloadmode SL_MANUAL. Together these disallow calling other shared libraries unless they are registered already.

Error LME_REENTER can be prevented by the following restrictions. LM_LOADCO, LM_UNLOADCO, and shared library entry functions called with events SL_REGEV or SL_UNREGEV should not call other shared libraries, sl_register(), sl_acquire(), sl_unregister(), or sl_release(). Shared library entry functions called with SL_ATTACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire(). Shared library functions called with SL_DETACHEV should not call the before-mentioned system calls. However, they can call other shared libraries if and only if all shared libraries are registered by sl_register() or sl_bindindex() before any of them are acquired by sl_acquire() and all of

the shared libraries have unloadmode SL_MANUAL so they are not unregistered by sl_release(). If a shared library entry function called with SL_REGEV or SL_ATTACHEV needs to call other shared libraries and the above restrictions cannot be met, the entry function can be written to set a flag in the shared library's data area, indicating that a register or attach needs to be done. Then, all of the functions exported by the shared library would check these flags and do the deferred register or attach initialization before their normal processing.

All the pLM+ functions are designed for a multitasking environment. Concurrent pLM+ system calls wait, if necessary, to maintain consistency or provide a sequential ordering of pLM+ system calls.

Arguments

index Points to the index of the shared library.

Return Value

This function returns a 0 on success, and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

1. Many errors can occur during the unregistering of dependent libraries such as entry function errors, unregister callout errors, etc. Also, more than one error condition can occur during one sl_unregister() call, as sl_unregister() ignores the errors and continues to unregister all the acquired unique dependent libraries of the root library corresponding to index, to avoid orphan libraries. sl_unregister() returns the error that is encountered first.
2. LM_UNLOADCO does not necessarily unload anything. If the shared library is part of the pSOSystem image, LM_UNLOADCO returns without unloading anything. On the other hand, if the shared library is not part of the pSOSystem image, LM_LOADCO does actually unload the shared library. (See "Loader" in Chapter 1, "System Services" in the *pSOSystem Programmer's Reference* manual.)
3. sl_unregister() can be used without any LM_UNLOADCO at all if none of the shared libraries has unloadmode SL_AUTOUNLOAD.

See Also

sl_register

sl_update

Replaces a currently registered library with another library with the same name.

```
#include <plm.h>
#include <plmcfg.h>
unsigned long sl_update(
   unsigned long index, /* index of library to be replaced */
   const sl_header *libaddr,/* new library header address */
   const sl_attrib *attr    /* attributes of new library
   )
```

Description

The `sl_update()` function verifies if there is a legal registered library at `index` in the pLM+'s table. If there is one, it reads the dispatch header of the new library pointed by `libaddr` and verifies that both the libraries have the same name. If they have the same name, then `sl_update()` places the new library header address `libaddr` at the same entry pointed by `index` in the pLM+'s table, replacing the current library. The dispatch header pointed by `libaddr` must be in the proper format and should not be modified.

If the current library has been acquired by some task before `sl_update()` is called, then `sl_update()` also checks if all the names of the first level dependent libraries of both the current and new libraries as stored in the library headers are the same, before replacing the current library. Only if they are the same, it replaces the current library as above. The reference count of number of tasks attached to the new library is set to be the same as the old library. Per task acquisition count of the new library is also set to be the same as that of the old library. Since the new library simply takes the place of the old library, `sl_update()` does not call the entry functions of the old and new libraries.

`sl_update()` does not load the new library into memory. The new library must already be present in memory prior to calling `sl_update()`. The attribute structure pointed at by `attr`, if present, as denoted by a non-zero value of `attr`, can be used to set the unloadmode and loadhandle of the new library. See `sl_register()` for different values of unloadmode. If `attr` has a non-zero value, `sl_update()` reads the unloadmode and loadhandle fields of this structure and sets these attributes of the library. It ignores all other fields of the attributes structure. If `attr` has a zero value, then it sets unloadmode to `SL_MANUAL` mode and loadhandle to zero. The unloadmode and loadhandle may also be set later on the new library by calling `sl_setattr()`. Also, `sl_update()` does not unload the old library. The old library's unloadmode is ignored and its loadhandle is considered lost. If one needs to

get the loadhandle of the old library, then `sl_getattr()` must be called on `index` before `sl_update()` is called.

If `sl_update()` is successful, all the future references to `index` as well as the library with the same name are directed to the new library. `sl_update()` is useful in upgrading the system with new versions of libraries without having to reboot the system.

Note that even though the tasks have acquired the current library requiring exact or compatible match of versions, `sl_update()` can allow an incompatible version to be placed at the same index. Also, the dependent libraries of the new library may have different version requirements than the currently acquired dependent libraries, hence those libraries may now require updates. It is the programmer's responsibility to perform such additional updates as well as to handle any effects due to version conflicts.

Also, note that at the time `sl_update()` is called, some tasks may be still executing within the current library (calls in progress) and also may have cached function pointers pointing to the current library. The tasks may continue to use the current library even after it is replaced by the new library. It is the system programmer's responsibility to handle any effects of the same program using two versions of a library at the same time and to decide when to unload the old library from memory. Also, it is the programmer's responsibility to handle the static and global data in the old library. Any modified static and global data in the old library are not copied to the new library and hence are considered lost. Since, `sl_update()` does not call the entry functions of new and old libraries, it is the programmer's responsibility to call them appropriately. It is recommended that the `sl_update()` operation is well planned on a system wide basis.

Arguments

`index`	Index of library to be replaced.
`libaddr`	New library header address.
`attr`	Attributes of the new library.

Return Value

This function returns a 0 on success, and an error code on failure.

Error Codes

Refer to Appendix A.

Notes

None.

See Also

sl_register, *sl_acquire*

6 pNA+ System Calls

This chapter provides detailed information on each system call in the pNA+ component of pSOSystem. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, a detailed description, its arguments, its return value, and any error codes that it can return. Where applicable, the section also includes the headings "Notes," "Usage," and "See Also." "Notes" contains any important information not specifically related to the call description; "Usage" provides detailed usage information; and "See Also" indicates other calls that have related information.

Structures described in this chapter are also defined in the file `<pna.h>`. Structures must be word-aligned and must not be packed.

If you need to look up a system call by its functionality, refer to Table 6-1 on page 6-2, which lists the calls alphabetically by component and provides a brief description of each call.

For more information on error codes, refer to Appendix A, *Error Codes*, which lists the codes numerically and shows which pNA+ System calls are associated with each one.

NOTE: All pNA+ system calls return an `int` value. Previous releases of pSOSystem lower than 2.5 returned a `long` value. This is done to retain compatibility with Berkeley socket API. Similarly, change all the socket calls to accept the structure `sockaddr`, instead of, `sockaddr_in` (which was used in pSOSystem versions lower than 2.5).

TABLE 6-1 pNA+ System Calls

Name	Description	Page
accept	Accepts a connection on a socket.	6-5
add_ni	Adds a network interface.	6-7
bind	Binds an address to a socket.	6-9
close	Closes a socket descriptor.	6-11
connect	Initiates a connection on a socket.	6-12
get_id	Gets a task's user ID and group ID.	6-14
getpeername	Gets the address of a connected peer.	6-15
getsockname	Gets the address that is bound to a socket.	6-17
getsockopt	Gets options on a socket.	6-19
ioctl	Performs control operations on a socket.	6-23
listen	Listens for connections on a socket.	6-41
pna_allocb	Allocates a message block.	6-42
pna_esballoc	Attaches a message block to the data buffer.	6-44
pna_freeb	Frees a message block.	6-46
pna_freemsg	Frees all the message blocks associated with a message.	6-47
recv	Receives data from a socket.	6-48
recvfrom	Receives data from a socket.	6-51
recvmsg	Receives data from a socket.	6-54
select	Checks the status of multiple sockets.	6-57
send	Sends data to a socket.	6-60
sendmsg	Sends data to a socket.	6-62
sendto	Sends data to a socket.	6-64
set_id	Sets a task's user ID and group ID.	6-67
setsockopt	Sets options on a socket.	6-68

TABLE 6-1 pNA+ System Calls (Continued)

Name	Description	Page
shr_socket	Obtains a new socket descriptor for an existing socket.	6-75
shutdown	Terminates all or part of a full-duplex connection.	6-76
socket	Creates a socket.	6-77

accept

Accepts a connection on a socket.

```
#include <pna.h>
long accept(
   int s,                        /* socket descriptor */
   struct sockaddr *addr,        /* socket structure */
   int *addrlen                  /* socket structure size */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

```
-D_PNA_30_BACK
```

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

This call is used to accept a connection request that the specified socket receives from a foreign socket. Servers use `accept()` with connection-oriented or stream (TCP) sockets.

Before `accept()` is called, the socket must be set up to receive a connection request by issuing the `listen()` system call. `accept()` extracts the first connection request on the queue of pending connections; creates a new socket with the same properties as the original socket; completes the connection between the foreign socket and the new socket; and returns a descriptor for the new socket. The new returned socket descriptor is used to read from and write data to the foreign socket. It is not used to accept more connections. The original socket remains open for accepting further connections.

If no pending connections exist on the queue and the socket is not marked as non-blocking, `accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept()` returns an error.

Upon return, `accept()` stores the address of the connected socket in the specified socket address structure.

Arguments

s	Specifies the socket on which to accept a connection request.

addr	Points to a structure of type sockaddr_in where accept() stores the address of the connected socket. The structure sockaddr_in is defined in the file <pna.h> and has the following format:
	`struct sockaddr_in {`
	`  short sin_family;  /* must be AF_INET */`
	`  unsigned short sin_port;  /* 16-bit port number */`
	`  struct in_addr sin_addr;  /* 32-bit IP address */`
	`  char sin_zero[8];  /* must be 0 */`
	`  };`
	This structure cannot be packed.

addrlen	Points to an integer equal to 16, which is the size in bytes of the sockaddr_in structure.

Return Value

If this call succeeds, it returns a non-negative integer that is a descriptor for the accepted socket. It returns -1 on error.

Error Codes

Refer to Appendix A.

See Also

bind, *connect*, *listen*, *select*, *socket*

add_ni

Adds a network interface.

```
#include <pna.h>
long add_ni(
   struct ni_init *ni   /* network interface */
   )
```

Description

This system call is used to dynamically add a network interface to the pNA+ network manager. The characteristics of the network interface are specified in the data structure pointed to by ni.

This routine calls the network interface driver's NI_INIT routine for driver initialization.

Arguments

ni — Points to an ni_init structure. The structure ni_init is defined in the file <pna.h> and has the following format:

```
struct ni_init {
   int (*entry)();   /* address of NI entry point */
   int ipadd;        /* IP address */
   int mtu;          /* maximum transmission length */
   int hwalen;       /* length of hardware address */
   int flags;        /* interface flags */
   int subnetaddr;   /* subnet mask */
   int dstipaddr;    /* Dest. address in Point-to Point NI */
   int reserved[1];  /* reserved for future use */
   };
```

This structure cannot be packed. The flags element can contain one or more of the following symbolic constants (defined in pna.h), using the syntax:

```
IFF_BROADCAST | IFF_RAWMEM
```

Symbolic Constant	Description
IFF_BROADCAST	NI can broadcast.
IFF_EXTLOOPBACK	NI uses external loopback.
IFF_INITDOWN	NI must be initialized in DOWN state. Default is UP state.
IFF_MULTICAST	NI supports multicast.

IFF_NOARP	NI does not have ARP.
IFF_POINTTOPOINT	NI is point-to-point driver.
IFF_POLL	pNA+ polls NI at regular intervals
IFF_RAWMEM	NI passes packets as mblks.
IFF_UNNUMBERED	NI is an unnumbered link.

Return Value

This system call returns the pNA+ interface number of the new network interface if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

Network Interfaces and Configuration Tables, *pSOSystem Programmer's Reference*.

bind

Binds an address to a socket.

```
#include <pna.h>
long bind(
   int s,                        /* socket descriptor */
   struct sockaddr *addr,        /* socket structure */
   int addrlen                   /* structure size */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

`-D_PNA_30_BACK`

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

This system call is used to assign (or bind) an address (a 32-bit internet address and a 16-bit port number) to a socket. A socket is created without an address and cannot be used to receive data until it is assigned one. Raw IP sockets are an exception. If they are unbound then they receive all packets regardless of the packet's addresses.

To simplify address binding, a wildcard internet address is supported to free the user from needing to know the local internet address. It also makes programs more portable. When the internet address is specified as the symbolic constant `INADDR_ANY`, pNA+ interprets it as any valid address. This allows the socket to receive data regardless of its node's internet address. For example, if a socket is bound to `<INADDR_ANY, 10>` and it resides on a node that is attached to networks 90.0.0.2 and 100.0.0.3, the socket can receive data addressed to `<90.0.0.2, 10>` or `<100.0.0.3, 10>`.

Arguments

s	Specifies the socket to which the address is bound.
addr	Points to a structure of type sockaddr_in that stores the socket attributes to be bound to the socket. The structure sockaddr_in is defined in the file <pna.h> and has the following format:

```
struct sockaddr_in {
   short sin_family;            /* must be AF_INET */
   unsigned short sin_port;     /* 16-bit port number */
   struct in_addr sin_addr;     /* 32-bit IP address */
   char sin_zero[8];            /* must be 0 */
   };
```

This structure cannot be packed.

addrlen	Specifies the size in bytes of the sockaddr_in structure and must be 16.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

connect, *getsockname*, *listen*, *socket*

close

Closes a socket descriptor.

```
#include <pna.h>
long close(
   int s   /* socket descriptor */
   )
```

Description

The `close()` call discards the specified socket descriptor. If it is the last descriptor associated with the socket, the socket is deleted and, unless the `SO_LINGER` socket option is set (see *getsockopt* *on page 6-19* and *setsockopt* *on page 6-68*), any data queued at the socket is discarded.

As a special case, if the specified socket descriptor is equal to 0, `close()` closes all socket descriptors that have been allocated to the calling task.

Arguments

`s`	Specifies the socket to be closed.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

`socket`, *`setsockopt`*

connect

Initiates a connection on a socket.

```
#include <pna.h>
long connect(
   int s,                        /* socket descriptor */
   struct sockaddr *addr,        /* socket attributes */
   int addrlen                   /* attribute size */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

`-D_PNA_30_BACK`

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

This system call is used to establish an association between a local socket and a foreign socket.

Generally, a stream socket connects only once. A datagram socket can use `connect()` multiple times to change its association. A datagram socket can dissolve the association by connecting to an invalid address, such as the null address `INADDR_ANY` defined in `pna.h`.

If a stream socket is specified, `connect()` initiates a connection request to the foreign socket. The caller is blocked until a connection is established, unless the socket is non-blocking.

If a datagram socket is specified, `connect()` associates the socket with the socket address supplied. This address is used by future `send()` calls to determine the datagram's destination. This is the only address from which datagrams can be received.

If a raw socket is specified, `connect()` associates the socket with the socket address supplied. This address is used by future `send()` calls to determine the datagram's destination. This is the only address from which datagrams can be received.

Arguments

s	Specifies the local socket.

addr	Points to a sockaddr_in structure that contains the address of the foreign socket. The structure sockaddr_in is defined as follows:
	`struct sockaddr_in {`
	`  short sin_family;  /* must be AF_INET */`
	`  unsigned short sin_port;  /* 16-bit port number */`
	`  struct in_addr sin_addr;  /* 32-bit IP address */`
	`  char sin_zero[8];  /* must be 0 */`
	`  };`
	This structure cannot be packed.

addrlen	Specifies the size in bytes of the sockaddr_in structure and must be 16.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

accept, *close*, *connect*, *getsockname*, *select*, *socket*

get_id

Gets a task's user ID and group ID.

```
#include <pna.h>
long get_id(
   long *userid,   /* task user ID */
   long *groupid,  /* task group ID */
   long *groups;   /* must be zero */
   )
```

Description

This system call obtains the user ID and group ID of the calling task. These IDs are used for accessing NFS servers. The user ID and group ID are set by using the `set_id()` call. Default values for these IDs are defined in the pNA+ Configuration Table.

Arguments

`userid`	Points to a long variable where `get_id()` stores the calling task's user ID.
`groupid`	Points to a long variable where `get_id()` stores the calling task's group ID.
`groups`	Zero must be passed as a third argument (which is currently ignored).

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

None.

See Also

set_id

getpeername Gets the address of a connected peer.

```
#include <pna.h>
long getpeername(
   int s,                        /* socket descriptor */
   struct sockaddr *addr,        /* socket attributes */
   int *addrlen                  /* socket structure size */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

```
-D_PNA_30_BACK
```

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

The `getpeername()` call obtains the address of the *peer* connected to the specified socket. The peer is the socket at the other end of the connection.

Arguments

`s` Specifies the original socket.

`addr` Points to a `sockaddr_in` structure in which `getpeername()` stores the address of the peer socket. The structure `sockaddr_in` is defined as follows:

```
struct sockaddr_in {
   short sin_family;             /* must be AF_INET */
   unsigned short sin_port;      /* 16-bit port number */
   struct in_addr sin_addr;      /* 32-bit IP address */
   char sin_zero[8];             /* must be 0 */
   };
```

This structure cannot be packed.

`addrlen` Points to an integer equal to 16, which is the size in bytes of the `sockaddr_in` structure.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

accept, *bind*, *getsockname*, *socket*

getsockname

Gets the address that is bound to a socket.

```
#include <pna.h>
long getsockname(
   int s,                        /* socket descriptor */
   struct sockaddr *addr,        /* socket attributes */
   int *addrlen                  /* size of sockaddr_in */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

```
-D_PNA_30_BACK
```

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

The `getsockname()` call obtains the address bound to the specified socket.

Arguments

`s`	Specifies the socket.

`addr`	Points to a `sockaddr_in` structure in which `getsockname()` stores the address bound to the socket. The `sockaddr_in` structure is defined as follows:
	`struct sockaddr_in {`
	`  short sin_family;  /* must be AF_INET */`
	`  unsigned short sin_port; /* 16-bit port number */`
	`  struct in_addr sin_addr; /* 32-bit IP address */`
	`  char sin_zero[8];  /* must be 0 */`
	`  };`
	`This structure cannot be packed.`

`addrlen`	Points to an integer equal to 16, which is the size in bytes of the `sockaddr_in` structure.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

bind, *getpeername*, *socket*

getsockopt

Gets options on a socket.

```
#include <pna.h>
long getsockopt(
   int s,       /* socket descriptor */
   int level,   /* SOL_SOCKET, IPPROTO_IP, */
                /* or IPPROTO_TCP */
   int optname, /* retrieval option */
   char *optval,/* return buffer */
   int *optlen  /* input/output buffer size */
   )
```


Description

The `getsockopt()` system call obtains the status of options associated with the specified socket. Socket level, IP protocol level, or TCP protocol level options may be retrieved.

Arguments

`s`	Specifies the socket.
`level`	Specifies the level of the option to be queried and must be set to `SOL_SOCKET` for socket level operations, `IPPROTO_IP` for IP protocol level operations, or `IPPROTO_TCP` for TCP protocol level operations.
`optname`	Specifies the option to be queried, and uses a symbolic constant. The symbolic constants available for each `level` are provided below and in `<pna.h>`.
`optval`	Points to a buffer where `getsockopt()` stores the value for the requested option. For most options, an `int` is returned in the buffer pointed to by `optval`. A nonzero value means the option is set, and a 0 means the option is off.
`optlen`	An input-output parameter. On input, it should contain the size of the buffer pointed to by `optval`. On output, it contains the actual size of the value returned in the `optval` buffer.

Socket Level Options

`level` must be set to `SOL_SOCKET` for socket level operations. The `optname` value can be one of the following:

`SO_BROADCAST` Allows broadcast datagrams on a socket.

`SO_DONTROUTE` Indicates that the outgoing messages should not be routed. Packets directed to unconnected networks are dropped.

`SO_ERROR` Returns the pending error and clears the error status.

`SO_KEEPALIVE` Keeps the connection alive by periodically transmitting a packet over socket `s`.

`SO_LINGER` Controls the action taken when unsent messages are queued on a socket and a `close()` is executed. If the socket is a stream socket and `SO_LINGER` is set (`l_onoff` set to 1), the calling task blocks until it can transmit the data or until a timeout period (linger time, in seconds) expires. If `SO_LINGER` is disabled (`l_onoff` set to 0), the socket is deleted immediately. `SO_LINGER` uses the `linger` structure, which is defined as follows:

```
struct linger {
   int l_onoff;    /* on/off option */
   int l_linger;   /* linger time in secs.*/
   }
```

`This structure cannot be packed.`

`SO_OOBINLINE` Requests that out-of-band data go into the normal data input queue as received; it then is accessible with `recv()` calls without the `MSG_OOB` flag.

`SO_RCVBUF` Adjusts the normal buffer size allocated for a socket input buffer.

`SO_REUSEADDR` Indicates that local addresses can be reused in a `bind()` call.

`SO_REUSEPORT` Indicates that local addresses can be reused in a `bind()` call. For more information, see section 4.4.3 of the Network Programming chapter in *pSOSystem System Concepts*.

`SO_SNDBUF` Adjusts the normal buffer size allocated for a socket output buffer.

`SO_TYPE` Returns the type of socket.

TCP Level Option

level must be set to IPPROTO_TCP for TCP protocol level operations. The argument optname can have the following value:

TCP_KEEPALIVE_CNT	Number of Keepalive strobes. Upon expiration of the Keepalive idle timer TCP will send a number of strobes separated by a fixed interval. If the other end fails to respond to the strobes (special TCP segments) then the TCP connection will be terminated. The default number of strobes in pNA+ are 8. Only valid if the SO_KEEPALIVE option is set above.
TCP_KEEPALIVE_IDLE	Keepalive idle time in TCP. If the connection has been idle for this time, the timer will expire causing TCP to send a special segment forcing the other end to respond. On demand-dial links for example the timer may be set long enough so as not to cause unnecessary traffic. The default in pNA+ is 120 seconds (2 hrs). The timer is in seconds. Only valid if the SO_KEEPALIVE option is set above.
TCP_KEEPALIVE_INTVL	Keepalive strobe interval. The strobes sent out by TCP upon expiration of the Keepalive idle timer are separated by a fixed interval. The default interval between the strobes is set to 75 seconds in pNA+. The timer is in seconds. Only valid if the SO_KEEPALIVE option is set above.
TCP_MSL	Maximum Segment Lifetime in TCP. This controls the TIME_WAIT or 2MSL timer in TCP which is set to twice the value of the MSL. The timer is used to validate connection termination and transmits remaining data in the send queue. It is a safeguard against sequence numbers being overlapped. If set to a low value it allows the sockets to be quickly deleted. The default in pNA+ is 30 seconds. The timer is in seconds.
TCP_NODELAY	Disables delay acknowledgment algorithm. Data is sent immediately over the network instead of waiting for the window to be full.

IP Level Options

`level` must be set to `IPPROTO_IP for IP protocol level operations`. The argument `optname` can have one of the following values:

`IP_HDRINCL`	Specifies that the IP Header will be included in the output packets. The following fields will be set by pNA+ if they are set to 0 in the included IP header: IP identification number and IP source address. The fragmentation offset and checksum fields are always computed by pNA+. The rest of the IP header fields must be set appropriately.
`IP_MULTICAST_IF`	Specifies the outgoing interface for multicast packets. For this option, `optval` is a pointer to `struct in_addr`. If set to `INADDR_ANY`, then the routing table will be used to select an appropriate interface.
`IP_MULTICAST_INTF`	Specifies the outgoing interface for multicast packets, the interface defined by its interface number. The `optval` is a pointer to a long. If set to -1 (default), the routing table will be used to select an appropriate interface.
`IP_MULTICAST_LOOP`	Specifies whether or not to loop back multicast packets. `optval` is a pointer to an unsigned char. A value of 0 disables loopback. By default, the packets are looped back (`IP_DEFAULT_MULTICAST_LOOP`.)
`IP_MULTICAST_TTL`	Specifies the time-to-live for outgoing IP multicast datagrams. `optval` is a pointer to an unsigned char. The default is `IP_DEFAULT_MULTICAST_TTL`.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

`setsockopt`, *`socket`*

ioctl

Performs control operations on a socket.

```
#include <pna.h>
long ioctl(
   int s,      /* socket descriptor */
   int cmd,    /* socket operation */
   char *arg   /* operation argument */
   )
```

Description

The `ioctl()` system call is used to perform control operations on the specified socket.

Arguments

`s`	Specifies the socket.
`cmd`	Specifies the operation and is a symbolic constant. All permissible `cmd` values, except MIB-related operations, are defined in `<pna.h>`. MIB-related `cmd` values are defined in `<pna_mib.h>`. Operation descriptions are provided below.
`arg`	Points to a data structure that is dependent on the value of `cmd` and contains additional information needed by `ioctl()` to perform the operation.

Operations

System-Related Operations

Operation	Description
`FIOASYNC`	Controls whether or not the user-provided signal handler is called when events related to the socket occur (for example, if the socket receives urgent data). If the integer pointed to by `arg` equals 1, signalling is enabled. If the integer pointed by `arg` equals 0, signalling is disabled.
`FIOGETOWN`	Identifies the owner of the socket. The task ID of the owner task is returned in the integer variable pointed to by `arg`.

Operation	Description
FIONBIO	Sets the blocking mode of the socket. If the integer pointed to by arg equals 1, the socket is set to operate in non-blocking mode. If the integer pointed to by arg equals 0, the socket is set to operate in blocking (default) mode. Normally, socket operations that cannot be immediately completed cause the task that initiated the operation to block. If a socket is marked non-blocking, an operation request that cannot complete without waiting does not execute, and an error is returned.
FIOREAD	Returns the number of bytes stored in the socket's input buffer in the integer pointed to by arg.
FIOSETOWN	Assigns an owner to the socket. The parameter arg should point to an integer that provides the task ID (tid) of the socket's owner. This tid is passed as an input parameter to the user signal handler.
SIOCATMARK	Determines whether out-of-band is available. If the data available at the socket is out-of-band data, the integer pointed to by arg is set equal to 1. Otherwise, it equals 0 upon return.
SIOCSSBMAX	Assigns a maximum total default size of 128 Kbytes for send and receive socket buffer queues upon creation of a socket. This operation changes the maximum total size. This may be used to increase end-to-end throughput for fast networks. The value is an integer pointed to by arg.
SIOCGSBMAX	Gets the total maximum send and receive socket buffer queue size that pNA+ assigns to newly created sockets. It is returned by the integer variable pointed to by arg.

NI-Related Operations

The following operations are available to access or modify the characteristics of a Network Interface:

Operation	Description
SIOCSIFADDR	Sets the interface address.
SIOCGIFADDR	Gets the interface address.
SIOCSIFBRDADDR	Sets the IP broadcast address of the NI.
SIOCGIFBRDADDR	Gets the IP broadcast address of the NI.

Operation	Description
SIOCSIFDSTADDR	Sets point-to-point address for the interface.
SIOCGIFDSTADDR	Gets point-to-point address for the interface.
SIOCSIFMTU	Sets the maximum transmission unit of the NI.
SIOCGIFMTU	Gets the maximum transmission unit of the NI.
SIOCSIFNETMASK	Sets the network mask.
SIOCGIFNETMASK	Gets the network mask.
SIOCSIFFLAGS	Sets the interface flags field. If the interface is marked down, any packets currently routed through the interface are re-routed or dropped, resulting in a send error condition. IFF_POLL, IFF_EXTLOOPBACK and IFF_UP flags can be set by using this call.
SIOCGIFFLAGS	Gets interface flags.
SIOCGIFCONF	Gets the interface configuration list. When this command is used, arg must point to an ifconf structure (see the structure below, and the test case in Example 6-1). The ifc_len field initially should be set to the buffer size pointed to by ifc_buf. On return, ifc_len has the configuration list length in bytes.


```
/*
* Structure used in SIOCGIFCONF request.
* Used to retrieve interface configuration
* for machine (useful for programs that must
* know all accessible networks.)
*/
struct ifconf {
   int ifc_len;      /* size of associated buffer */
   union {
      char *ifcu_buf;
      struct ifreq *ifcu_req;
   } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf  /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req  /* array of structures returned */
};
```

EXAMPLE 6-1: Example Test Case for SIOCGIFCONF

```
struct ifconf ifconf;{
int num_ifs = MAX_IF, len;
struct ifreq *ifreqs = (struct ifreq *)NULL;

len = sizeof(struct ifreq) * num_ifs;
if ((ifreqs = (struct ifreq *)malloc(len)) == NULL)
```

```
return;

memset(ifreqs, 0, len);
ifconf.ifc_len = len;
ifconf.ifc_req = ifreqs;
if (ioctl(sock, SIOCGIFCONF, (caddr_t)&ifconf) < 0) {
free(ifreqs);
return;
}
```

For all other NI-related operations, `arg` must point to the following structure:

```
struct ifreq {
   long ifr_ifno;   /* Interface number of the NI */
   union {
      struct sockaddr ifru_addr;     /* IP address of the NI */
      struct sockaddr ifru_dstaddr;  /* Dest addr p-to-p link */
      struct sockaddr ifru_broadaddr;/* NI broadcast address */
      unsigned long ifru_flags;      /* Flags for the NI */
      int ifru_mtu;                  /* Maximum number of
                                      * transmission units */
      char *ifru_data;               /* For private use */
      } ifr_ifru;
   #define ifr_addr   ifr_ifru.ifru_addr        /* Address */
   #define ifr_dstaddr    ifr_ifru.ifru_dstaddr /* Other end of */
                                                /* p-to-p link */
   #define ifr_broadaddr ifr_ifru.ifru_broadaddr/* NI broadcast addr. */
   #define ifr_mtu ifr_ifru.ifru_mtu            /* Maximum number of
                                                 * transmission units */
   #define ifr_data ifr_ifru.ifru_data       /* NI private data */
   #define ifr_flags ifr_ifru.ifru_flags     /* Flags */
};
```

`ifr_flags` can contain one or more of the symbolic constants below (defined in `pna.h`), in the following syntax:

```
IFF_BROADCAST | IFF_RAWMEM
```

Symbolic Constant	Description
IFF_BROADCAST	NI can broadcast.
IFF_EXTLOOPBACK	NI uses external loopback.
IFF_INITDOWN	Initialize an interface in the DOWN state. By default interfaces are installed in the UP state. This flag may only be specified when initially adding an interface. Note that this is not an attribute of the NI.
IFF_MULTICAST	NI supports multicast.
IFF_NOARP	NI does not have ARP.
IFF_POINTTOPOINT	NI is point-to-point driver.

Symbolic Constant	Description
IFF_POLL	pNA+ polls NI at regular intervals.
IFF_RAWMEM	NI passes packets as mblks.
IFF_UNNUMBERED	NI is an unnumbered link.
IFF_UP	NI is UP.
IFF_PROMISC	NI is running in promiscuous mode.
IFF_RECEIVEOFF	NI receiver is disabled. Packets are not received in this mode.

Memory-Related Operations

SIOCGMBSTAT Gets statistics for *mblks* (memory blocks) configured in the system. A pointer to the `mbstat` structure is passed via the `arg` parameter of the `ioctl()` call. `mbstat` is defined below:

```
struct mbstat {
   long mb_bufclasses;   /* Number of buffer classes */
   long mb_mblks;     /* Number of mblks configured */
   long mb_free;      /* Number of free mblks */
   long mb_wait;      /* Number of times task waited for mblk */
   long mb_drops;     /* Number of times mblks unavailable */
};
```

SIOCGDBSTAT

- Gets the statistics for buffers configured in the system. A pointer to the `dbreq` structure is passed as the `arg` parameter. The `dbs` element must point to a buffer that can hold at least `size` number of bytes. The buffer on return contains a sequence of dbstat structures each of which is filled in the statistics of the particular buffer size. It is recommended that the size of the `dbs` buffer be large enough to contain all the buffer types configured in the system. `size` is an input/output element that contains the size of a buffer on input. On output the pNA+ network manager returns the size of buffer used by the call. These structures are defined as follows in `pna.h`:

```
struct dbreq {
   long size;            /* Size of the buffer */
   struct dbstat *dsp;   /* Pointer to the buffer*/
};

struct dbstat {
   long db_size;       /* Size of buffer */
   long db_nbuffers;   /* Number of buffers configured*/
   long db_free;       /* Number of free buffers */
   long db_wait;       /* No. of times tasks waited for buffer */
   long db_drops;      /* No. of times buffer was unavailable */
};
```

ARP-Related Operations

The following operations are available for accessing and modifying the ARP table:

Operation	Description
SIOCSARP	Sets an ARP entry.
SIOCGARP	Gets an ARP entry.
SIOCDARP	Deletes an ARP entry.

For all ARP-related operations, arg must point to the following structure:

```
struct arpreq {
   struct sockaddr arp_pa;   /* protocol address */
   struct sockaddr arp_ha;   /* hardware address */
   int arp_flags;            /* flags */
};
```

The arp_pa structure is used to specify the host's IP address. The address family field in arp_pa must be AF_INET. The arp_ha structure is used to specify the host's hardware address. The arp_ha address field must be AF_UNSPEC.

arp_flags must be one of the following symbolic constants, defined in pna.h:

Symbolic Constant	Description
ATF_PERM	Permanent entry.
ATF_PUBL	Publish (respond for other host.)

ATF_PERM makes the entry permanent if the ioctl() call succeeds. ATF_PUBL specifies that ARP protocol should respond to ARP requests coming from other machines for the indicated host. This lets a host act as an ARP server, which can be useful in getting an ARP-only machine to talk to a non-ARP machine (Proxy ARP).

Routing-Related Operations

The following operations are available for manipulating the Routing Table:

Operation	Description
SIOCADDRT	Adds a routing table entry.
SIOCDELRT	Deletes a routing table entry.
SIOCMODRT	Modifies a routing table entry.

For all Routing-related operations, arg must point to struct rtentry. If a subnet mask must be specified with the route, the rt_flags field must have the RTF_MASK flag set and the rt_netmask field must be filled with the subnet mask.

The interface number of the route's output interface can be specified. Usually this is computed internally. But for unnumbered point-point links, for instance, you could specify an interface. The rt_ifno field is ignored unless the RTF_INTF flag is set.

```
struct rtentry {
   struct sockaddr rt_dst;       /* Specifies the destination
                                  * IP address of the route. */
   struct sockaddr rt_gateway;   /* Specifies the gateway
                                  * IP address for the route. */
   unsigned short rt_flags;      /* Specifies the type of route*/
   unsigned short reserved;      /* Reserved */
   unsigned long rt_netmask;     /* netmask of route */
   long rt_ifno;                 /* interface number */
   struct sockaddr *rt_prev_gateway;/* If multiple valid gateways
                                       are inserted to the same
                                       destination, the
                                       modification of the route
                                       requires both the new value
                                       for the gateway, and the
                                       previous value where it is
                                       modified to be specified
                                    here. */
   unsigned long reserved2;      /* Reserved */
};
```

The rt_flags element can contain one or more of the symbolic constants below (defined in pna.h), in the following syntax:

```
RTF_MASK | RTF_GATEWAY
```

Symbolic Constant	Description
RTF_HOST	Host entry (net otherwise).
RTF_GATEWAY	Destination is a gateway.
RTF_UP	Route is usable.
RTF_DYNAMIC	Route is added dynamically using the ICMP Redirect message.
RTF_MODIFIED	Route is modified using the ICMP Redirect message.
RTF_INTF	Route includes intf num.
RTF_MASK	Route includes subnet mask.

Network Node ID Operations

The following operations are available for accessing the pNA+ Network Node ID. This is also known as the Router ID. The Network Node ID may be equal to any one of the IP addresses assigned to the node. To set or get the Network Node ID, `arg` must point to an `in_addr` structure containing the IP address.

Operation	Description
`SIOCSNNODEID`	Sets the Network Node ID.
`SIOCGNNODEID`	Gets the Network Node ID.

Setting the IP TTL

Each IP packet sent by pNA+ (TCP/UDP or Raw IP) is assigned a TTL value. pNA+ assigns a default TTL value of 64 for outgoing UDP and TCP packets as per RFC 1700. Note that this system wide default value affects all sockets. The system wide default TTL value may be accessed by the IP group MIB commands `SIOCGIPDEFAULTTTL` and `SIOCSIPDEFAULTTTL`.

ICMP and Raw IP packets are assigned a fixed default TTL value for 255. This system wide default TTL value cannot be changed.

For UDP/IP multicast packets the default TTL value is defined to be 1 but may be modified using the `setsockopt()` call.

The following operations are available to change the TTL value on a per socket/connection basis. Initially the per socket TTL is set as per the rules above. The TTL value may be changed for each socket. To set or get the IP TTL value the `arg` parameter must point to an integer. The TTL value must be non-negative.

Operation	Description
`SIOCSIPTTL`	Sets the IP TTL value of the socket.
`SIOCGIPTTL`	Gets the IP TTL value of the socket.

UDP Checksum Operations

The following operations may be used to access or modify the UDP checksum computation status. By default, UDP checksum is not computed for outgoing UDP pack-

ets. The `arg` parameter must point to an integer. The integer is set to 1 to enable UDP checksum computation or 0 to disable the computation.

Operation	Description
SIOCSUDPCHKSUM	Sets the UDP checksum computation flag.
SIOCGUDPCHKSUM	Gets the UDP checksum computation flag.

MIB-II Related Operations

The `ioctl()` call is used to access pNA+ MIB-II objects, defined in this subsection. Refer to *pSOSystem System Concepts* for more details on `set` and `get` operations.

The operations described in the remainder of the `ioctl()` call description are defined by symbolic constants in `<pna_mib.h>`:

Definitions for Interface Group MIB Variables

GET Command Definitions

SIOCGIFNUMBER	Total number of interfaces
SIOCGIFTABLE	pNA+ Network Interface table
SIOCGIFDESCR	Description of NI
SIOCGIFTYPE	NI type
SIOCGIFMTUNIT	NI maximum transmission unit (mtu)
SIOCGIFSPEED	NI speed
SIOCGIFPHYSADDRESS	NI physical address
SIOCGIFADMINSTATUS	NI administration status
SIOCGIFOPERSTATUS	NI operational status
SIOCGIFLASTCHANGE	Last change in status of the NI
SIOCGIFINOCTETS	Number of octets received by the NI
SIOCGIFINUCASTPKTS	Number of unicast packets received by the NI
SIOCGIFINNUCASTPKTS	Number of multicast packets received by the NI
SIOCGIFINDISCARDS	Number of packets discarded by the NI
SIOCGIFINERRORS	Number of error packets received by the NI

GET Command Definitions

SIOCGIFINUNKNOWNPROTOS	Number of packets discarded by the NI due to unknown protocols
SIOCGIFOUTOCTETS	Number of octets transmitted by the NI
SIOCGIFOUTUCASTPKTS	Number of unicast packets sent by the NI
SIOCGIFOUTNUCASTPKTS	Number of non-unicast packets sent by the NI
SIOCGIFOUTDISCARDS	Number of outbound packets discarded by the NI
SIOCGIFOUTERRORS	Number of outbound packets discarded by the NI due to errors
SIOCGIFOUTQLEN	Length of output packet queue of the NI
SIOCGIFSPECIFIC	NI-specific parameter

SET Command Definitions

SIOCSIFADMINSTATUS	Set interface administration status of the NI

Note that in all of the interface group commands (except SIOCGIFTABLE), the arg should point to the struct mib_ifreq as follows:

```
struct mib_ifreq
  {
  long   ie_iindex;             /* Interface number (mib index) */
  union
      {
  long   ieu_index;          /* Interface number */
  char   *ieu_descr;         /* description of the interface */
  long   ieu_type;           /* type of the interface */
  long   ieu_mtu;            /* Maximum transmission unit */
  long   ieu_speed;          /* speed of the interface */
  struct sockaddr ieu_physaddress; /* media-specific address */
  long   ieu_adminstatus;    /* desired interface state */
  long   ieu_operstatus;     /* current operational status */
  long   ieu_lastchange;     /* last change in the interface status */
  long   ieu_inoctets;       /* total octets received from media */
  long   ieu_inucastpkts;    /* unicast packets delivered above */
  long   ieu_innucastpkts;   /* broadcast/muticast pkts delivered above */
  long   ieu_indiscards;     /* packets discarded due to resource limit */
  long   ieu_inerrors;       /* packets discarded due to format errors */
  long   ieu_inunknownprotos; /* packets for unknown protocols */
  long   ieu_outoctets;      /* total octets sent on the media */
  long   ieu_outucastpkts;   /* unicast packets from above */
```

```
  long   ieu_outnucastpkts; /* broadcast/multicast packets from above */
  long   ieu_outdiscards;   /* packets discarded due to res. limit */
  long   ieu_outerrors;     /* packets discarded due to errors */
  long   ieu_outqlen;       /* size of output queue */
  char   *ieu_specific;     /* MIB-specific pointer */
  } ieu_ieu;
};
```

For SIOCGIFTABLE, the arg should point to the struct mib_args, which is as follows:

```
struct mib_args
    {
    long len;
    char  *buffer;
    };
```

Definitions for IP Group MIB Variables

GET Command Definitions

SIOCGIPFORWARDING	IP gateway indication variable
SIOCGIPDEFAULTTTL	IP header default time-to-live value
SIOCGIPINRECEIVES	Input datagrams received from interfaces
SIOCGIPINHDRERRORS	Drops due to format errors
SIOCGIPINADDRERRORS	Drops due to invalid addresses
SIOCGIPFORWDATAGRAMS	IP datagrams forwarded
SIOCGIPINUNKNOWNPROTOS	IP datagrams discarded due to unknown protocol
SIOCGIPINDISCARDS	Input datagrams discarded with no problems
SIOCGIPINDELIVERS	Datagrams delivered to IP user-protocols
SIOCGIPOUTREQUESTS	Datagrams supplied by IP user-protocols
SIOCGIPOUTDISCARDS	Outbound datagrams discarded
SIOCGIPOUTNOROUTES	IP datagrams dropped due to no routes
SIOCGIPREASMTIMEOUT	IP reassembly queue timeout
SIOCGIPREASMREQDS	IP fragments needing reassembly
SIOCGIPREASMOKS	IP fragments reassembled
SIOCGIPREASMFAILS	IP fragments reassembly failures

GET Command Definitions

SIOCGIPFRAGOKS	IP datagrams successfully fragmented
SIOCGIPFRAGFAILS	IP datagram fragmentation failures
SIOCGIPFRAGCREATES	IP fragments created
SIOCGIPROUTINGDISCARDS	IP Routing entities discarded

SET Command Definitions

SIOCSIPFORWARDING	IP gateway indication variable
SIOCSIPDEFAULTTTL	IP header default time-to-live value
SIOCSIPREASMTIMEOUT	IP fragmentation reassembly queue timeout

Note that for all IP Group MIB variables, the `arg` points to the integer when the value is obtained. For corresponding set commands, it should point to the integer that contains the value to be set.

Definitions for IP NI Address Table

GET Command Definitions

SIOCGIPADDRTABLE	pNA+ NI IP address table
SIOCGIPADENTADDR	IP address of the NI
SIOCGIPADENTIFINDEX	Interface number of NI
SIOCGIPADENTNETMASK	Subnet mask of the NI
SIOCGIPADENTBCASTADDR	Broadcast address of the NI
SIOCGIPADENTREASMMAXSIZE	Maximum reassembly size of IP datagram

The structure used for the IP NI Address Table is the following:

```
struct mib_ipaddrreq
{
struct in_addr ia_iaddr;     /* IP address of this entry (mib index) */
  union
      {
      struct in_addr iau_addr;        /* IP address of this entry */
      long   iau_ifindex;       /* Interface number of the NI */
      long   iau_netmask;       /* subnet-mask for the IP address */
      long   iau_bcastaddr;     /* LSB of IP broadcast address */
      long   iau_reasmmaxsize;  /* Max. IP datagram reassembled */
      } iau_iau;
  };
```

Definitions for IP Route Table

GET Command Definitions

SIOCGIPROUTETABLE	IP routing table
SIOCGIPROUTEDEST	Route destination IP address
SIOCGIPROUTEIFINDEX	Interface number of the NI for the route
SIOCGIPROUTENEXTHOP	IP address of next hop of this route
SIOCGIPROUTETYPE	Type of this route
SIOCGIPROUTEPROTO	Protocol used by the route
SIOCGIPROUTEMASK	Network mask to be ANDed with destination address

SET Command Definitions

SIOCSIPROUTEDEST	Route destination IP address
SIOCSIPROUTENEXTHOP	IP addr of next hop of this route
SIOCSIPROUTETYPE	Type of this route

The structure used for IP Routing is the following:

```
struct mib_iproutereq
  {
     struct in_addr ir_idest;  /* Destination of the route (mib
                                  index) */
     union
     {
     struct in_addr iru_dest;    /* Destination of the route */
     long   iru_ifindex;         /* Interface number of the route */
     struct in_addr iru_nexthop; /* Gateway IP address for indirect
                                    address */
     long   iru_type;            /* Type of route */
     long   iru_proto;           /* mechanism used to determine
                                    route */
     long   iru_mask;            /* subnet-mask of the route */
   } iru_iru;
  };

#define ir_dest iru_iru.iru_dest
#define ir_ifindex iru_iru.iru_ifindex
#define ir_nexthop iru_iru.iru_nexthop
#define ir_type iru_iru.iru_type
#define ir_proto iru_iru.iru_proto
#define ir_mask iru_iru.iru_mask
```

Definitions for IP NET-TO-MEDIA Table

GET Command Definitions

SIOCGIPNETTOMEDIATABLE	IP Net-to-Media table
SIOCGIPNETTOMEDIAIFINDEX	Network Interface number of the NI for which the entry is valid
SIOCGIPNETTOMEDIAPHYSADDRESS	Physical address of this entry
SIOCGIPNETTOMEDIANETADDRESS	IP address of this entry
SIOCGIPNETTOMEDIATYPE	Type of this entry

SET Command Definitions

SIOCSIPNETTOMEDIAPHYSADDRESS	Physical address of this entry
SIOCSIPNETTOMEDIANETADDRESS	IP address of this entry
SIOCSIPNETTOMEDIATYPE	Type of this entry

For all of the IP NET-TO-MEDIA Table (except SIOCGIPNETTOMEDIATABLE), the arg is a pointer to the following structure:

```
struct mib_ipnettomediareq
  {
  struct in_addr inm_iaddr;     /* IP address of the mapping (mib index)*/
  long   inm_iifindex;          /* Interface number (mib index) */
  union
      {
      long   inmu_ifindex;      /* Interface number (mib index) */
      struct sockaddr inmu_physaddress; /* media address mapping */
      struct in_addr inmu_netaddress;   /* IP address of the mapping */
      long   inmu_type;         /* procedure mapping was determined by */
      } inmu_inmu;
  };
```

For the SIOCGIPNETTOMEDIATABLE, the arg is a pointer to the struct mib_args, which is as follows:

```
struct mib_args
    {
    long len;
    char  *buffer;
    };
```

Definitions for ICMP Group MIB Variables

GET Command Definitions

SIOCGICMPINMSGS	ICMP messages received
SIOCGICMPINERRORS	ICMP messages with format errors
SIOCGICMPINDESTUNREACHS	ICMP Destination Unreachable messages received
SIOCGICMPINTIMEEXCDS	ICMP Time Exceeded messages received
SIOCGICMPINPARAMPROBS	ICMP Parameter Problem messages received
SIOCGICMPINSRCQUENCHS	ICMP Source Quench messages received
SIOCGICMPINREDIRECTS	ICMP Redirect messages received
SIOCGICMPINECHOS	ICMP Echo (request) messages received
SIOCGICMPINECHOREPS	ICMP Echo Reply messages received
SIOCGICMPINTIMESTAMPS	ICMP Timestamp (request) messages received
SIOCGICMPINTIMESTAMPREPS	ICMP Timestamp Reply messages received

GET Command Definitions

SIOCGICMPINADDRMASKS	ICMP Address Mask Request messages received
SIOCGICMPINADDRMASKREPS	ICMP Address Mask Reply messages received
SIOCGICMPOUTMSGS	ICMP messages this entity sent
SIOCGICMPOUTERRORS	ICMP messages not sent due to ICMP problems
SIOCGICMPOUTDESTUNREACHS	ICMP Destination Unreachable messages sent
SIOCGICMPOUTTIMEEXCDS	ICMP Time Exceeded messages sent
SIOCGICMPOUTPARAMPROBS	ICMP Parameter Problem messages sent
SIOCGICMPOUTSRCQUENCHS	ICMP Source Quench messages sent
SIOCGICMPOUTREDIRECTS	ICMP Redirect messages sent
SIOCGICMPOUTECHOS	ICMP Echo (request) messages sent
SIOCGICMPOUTECHOREPS	ICMP Echo Reply messages sent
SIOCGICMPOUTTIMESTAMPS	ICMP Timestamp (request) messages sent
SIOCGICMPOUTTIMESTAMPREPS	ICMP Timestamp Reply messages sent
SIOCGICMPOUTADDRMASKS	ICMP Address Mask Request messages sent
SIOCGICMPOUTADDRMASKREPS	ICMP Address Mask Reply messages sent

For all ICMP MIB commands, the `arg` parameter points to an integer which either returns the value (for GET commands) or sets the new value (for SET commands).

Definitions for TCP Group MIB Variables

GET Command Definitions

SIOCGTCPRTOALGORITHM	TCP retransmission algorithm
SIOCGTCPRTOMIN	TCP minimum retransmission timeout

GET Command Definitions

SIOCGTCPRTOMAX	TCP maximum retransmission timeout
SIOCGTCPMAXCONN	TCP maximum simultaneous connections
SIOCGTCPACTIVEOPENS	Number of direct transitions to SYN-SENT state from the CLOSED state
SIOCGTCPPASSIVEOPENS	Number of direct transitions to SYN-RCVD state from the LISTEN state
SIOCGTCPATTEMPTFAILS	Number of failed TCP connection attempts
SIOCGTCPESTABRESETS	Number of TCP connections reset
SIOCGTCPCURRESTAB	Number of current TCP connections
SIOCGTCPINSEGS	Number of TCP segments received
SIOCGTCPOUTSEGS	Number of TCP segments sent
SIOCGTCPRETRANSSEGS	Number of TCP segments retransmitted
SIOCGTCPCONNTABLE	TCP connection table
SIOCGTCPCONNSTATE	State of this TCP connection
SIOCGTCPINERRS	Number of TCP segments received in error
SIOCGTCPOUTRSTS	Number of TCP segments sent with RST flag

For all TCP Group GET commands, the arg points to an integer that returns the value.

SET Command Definitions

SIOCSTCPCONNSTATE	State of this TCP connection

For SIOCSTCPCONNSTATE, arg points to struct mib_args. The buffer field in mib_args should point to the following structure:

```
struct mib_tcpconnreq
  {
  struct in_addr tc_localaddress; /* local IP address */
  long  tc_localport;             /* local port */
  struct in_addr tc_remaddress;   /* remote IP address */
  long tc_remport;                /* remote port */
  union
      {
      long tcu_state;             /* state of the connection */
      } tcu_tcu;
  };
```

Definitions for UDP MIB Variables

GET Command Definitions

SIOCGUDPINDATAGRAMS	UDP datagrams delivered to UDP users
SIOCGUDPNOPORTS	UDP datagrams received for unknown ports
SIOCGUDPINERRORS	UDP datagrams received with other errors
SIOCGUDPOUTDATAGRAMS	UDP datagrams sent from this entity
SIOCGUDPTABLE	pNA+ UDP listener table

For all UDP commands (except SIOCGUDPTABLE), the arg should point to an integer where the value is returned.

For SIOCGUDPTABLE, arg points to the struct mib_args. The buffer field in the mib_args structure should point to the following structure:

```
struct mib_udptabreq
    {
    struct in_addr u_localaddress; /* Local IP address */
    long u_localport;              /* Local port */
    };
```

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

socket, *setsockopt*, *getsockopt*

listen

Listens for connections on a socket.

```
#include <pna.h>
long listen(
   int s,         /* socket descriptor */
   int backlog    /* packet queue depth */
   )
```

Description

This system call sets up the specified socket to receive connections. Connection requests are queued on the socket until they are accepted with the `accept()` call. The maximum length of the queue of pending connections must be specified. If a connection request arrives while the queue is full, the requesting client gets an `ECONNREFUSED` error.

Arguments

`s`	Specifies the socket.
`backlog`	Defines the maximum length of the queue of pending connections.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

accept, *connect*, *socket*

pna_allocb

Allocates a message block.

```
#include <pna.h>
mblk_t *pna_allocb(size, pri)
int size,   /* Size of the data buffer */
int pri,    /* Memory partition */
)
```

Description

pna_allocb allocates a message block with a data buffer of a specified size.

The pNA+ memory manager searches the buffer list for the size that best fits the requested size. If a buffer of that size is not available, the call returns NULL. pna_allocb uses the following algorithm to find the best fit:

1. The pNA+ memory manager first searches for an exact match.
2. If a match is not available, the pNA+ memory manager searches for the smallest size able to contain the requested size.
3. If none is available, the maximum size configured in the pNA+ memory manager is used.

The following example illustrates this algorithm:

Let buffers of sizes 0, 128, 1024, and 4096 bytes be configured in pNA+. If a buffer of size 1024 is requested, the pNA+ memory manager allocates a buffer from the 1024-byte buffer list. A request for a 2048-byte buffer results in the pNA+ memory manager allocating a buffer from the 4096-byte buffer list, and a request for a size 8192 buffer results in the allocation of a 4096-byte buffer.

Arguments

size	Specifies the size of the data buffer.

`pri` Specifies the memory pool where memory is to be allocated from either transmit or receive.

The `pri` function parameter in the memory management routines `pna_allocb()` and `pna_esballoc()`are used to specify the memory resource pool where the allocation came from. The three memory resource pools are partitioned into:

- Data transmission resource allocation pool
- Data reception resource allocation pool
- pNA+ internal processing resource allocation pool

The application is allowed to specify either the data transmission resource allocation pool or the data reception resource allocation pool to request memory allocation. The pSOS+ system configuration file contains the specification on the actual amount of system memory allocation for each resource pool.

The possible values for the `pri` parameter are:

- `PNA_TXQ_MEMPOOL` for the transmission pool;
- `PNA_RXQ_MEMPOOL` for the reception pool.

If neither of these values is specified, the buffer is allocated from a transmit pool. If a centralized memory pool scheme is configured (with no separate pools for transmit and receive), this parameter is ignored.

Return Value

This system call returns a pointer to the message block if successful; otherwise, it returns a null pointer.

Error Codes

None.

See Also

`pna_esballoc`, *`pna_freeb`*, *`pna_freemsg`*

pna_esballoc

Attaches a message block to the data buffer.

```
#include <pna.h>
mblk_t *pna_esballoc(buffer, size, pri, frtn)
   unsigned char *buffer,   /* User-supplied data buffer */
   int size,                /* Size of buffer */
   int pri,                 /* Memory partition */
   frtn_t *frtn,            /* User free function */
)
```

Description

`pna_esballoc` allocates and attaches a message block to the user-supplied data buffer; it uses a zero-sized data block to attach the message block to the data buffer.

Arguments

`buffer` Points to the user-supplied data buffer.

`size` Specifies the size of `buffer`.

`pri` Specifies the memory pool where memory is to be allocated from either transmit or receive.

The `pri` function parameter in the memory management routines `pna_allocb()` and `pna_esballoc()`are used to specify the memory resource pool where the allocation came from. The three memory resource pools are partitioned into:

- Data transmission resource allocation pool
- Data reception resource allocation pool
- pNA+ internal processing resource allocation pool

The application is allowed to specify either the data transmission resource allocation pool or the data reception resource allocation pool to request memory allocation. The pSOS+ system configuration file contains the specification on the actual amount of system memory allocation for each resource pool.

The possible values for the `pri` parameter are:

- `PNA_TXQ_MEMPOOL` for the transmission pool;
- `PNA_RXQ_MEMPOOL` for the reception pool.

If neither of these values is specified, the buffer is allocated from a transmit pool. If a centralized memory pool scheme is configured (with no separate pools for transmit and receive), this parameter is ignored.

`frtn` Points to the `free_rtn` structure, which specifies a free routine and an argument to the free routine. The free routine is called by the pNA+ memory manager when the user-specified data buffer is being freed. The `free_rtn` structure is defined in `<pna.h>` as follows:

```
struct free_rtn {
   void (*free_func)();   /* User free routine */
   void *free_arg;        /* Argument to free routine */
};
typedef struct free_rtn frtn_t;
```


Return Value

This system call returns a pointer to the message block if successful; otherwise, it returns a null pointer.

Error Codes

None.

See Also

pna_allocb, *pna_freeb*, *pna_freemsg*

pna_freeb

Frees a message block.

```
#include <pna.h>
void pna_freeb (
   mblk_t *bp
)
```

Description

`pna_freeb()` deallocates a specified message block. This function decrements the reference to the data buffer. It then deallocates the data block and the associated data buffer when no more references to them exist. If the data buffer is user- supplied [see `pna_esballoc()`], the user-supplied free routine is called when no more references to the data buffer exist.

Arguments

bp	Points to the message block to be deallocated.

Return Value

None.

Error Codes

None.

See Also

pna_allocb, *pna_esballoc*, *pna_freemsg*

pna_freemsg

Frees all message blocks associated with a message.

```
#include <pna.h>
void pna_freemsg(bp)
   mblk_t *bp,   /* Points to the message to be freed */
)
```

Description

`pna_freemsg` frees all the message blocks and data blocks associated with the specified message. This routine calls the `pna_freeb()` routine to free individual message blocks.

Arguments

`bp`	Points to the message to be freed.

Return Value

None.

Error Codes

None.

See Also

`pna_allocb`, *`pna_esballoc`*, *`pna_freeb`*

recv

Receives data from a socket.

```
#include <pna.h>
long recv(
   int s,        /* socket descriptor */
   char *buf,    /* packet */
   int len,      /* packet length */
   int flags     /* packet attributes */
   )
```

Description

The `recv()` system call is used to receive data from the specified socket. The behavior of this system call depends on the socket type, as described under *Arguments*.

The `recv()` system call returns the number of bytes received, and this value should always be checked because this is the only way to detect the actual number of data bytes stored in the user buffer.

Applications can use this call to receive messages from the pNA+ network manager in a linked list of `mblks` (message blocks) by setting the `MSG_RAWMEM` flag. Using mblks eliminates the data copy performed in the pNA+ network manager during the data transfer between the application and the pNA+ network manager.

Arguments

`s` Specifies the socket from which data is received. `s` can be a stream, a datagram, or a raw socket.

If `s` is a stream socket, `recv()` copies whatever data is available at the socket to the user buffer and returns. `recv()` never copies more than `len` bytes of data to the user buffer, but it can copy less, if less than `len` bytes are available. Unless `ioctl()` was used to mark the socket non-blocking, `recv()` blocks the caller if no data is available at the socket. The caller is unblocked when data is received. If the socket has been marked non-blocking, `recv()` returns immediately whether or not data is received.

If `s` is a datagram socket, every `recv()` call receives one datagram. The sender defines the size of the datagram. If the `len` parameter is less than the size of the datagram, part of the datagram is discarded. The next `recv()` call reads the next datagram received but not the unread part of the previous datagram. Unless `ioctl()` was used to mark the socket non-blocking, `recv()` blocks the caller until a datagram is available at the socket. If the socket has been marked non-blocking, `recv()` returns immediately whether or not datagrams are received.

If `s` is a raw socket, every `recv()` call receives one raw datagram. The size of the raw datagram is defined by the sender. If the `len` parameter is less than the size of the raw datagram, part of the raw datagram is discarded. The next `recv()` call reads the next raw datagram received, not the unread part of the previous raw datagram. Unless `ioctl()` was used to mark the socket non-blocking, `recv()` blocks the caller until a raw datagram is available at the socket. If the socket has been marked non-blocking, `recv()` returns immediately whether or not raw datagrams are received. The packet contains an IP header along with the packet body, if any.

`buf` Points to the user buffer where the data is stored.

`len` Specifies the size in bytes of the buffer.

`flags` Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in <pna.h>). The default value of this parameter is 0.

- `MSG_OOB` Specifies that you want `recv()` to read any out-of-band data present on the socket, rather than the regular in-band data.
- `MSG_PEEK` Specifies that you want `recv()` to peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation sees the same data.
- `MSG_RAWMEM` Specifies that you have set `buf` to point to a linked list of mblks and `len` to the total size of the message.

Return Value

This system call returns either the number of bytes received or -1 if an error occurs. When the receive is shut down by either end of the connection, a value of 0 is returned.

Error Codes

Refer to Appendix A.

See Also

connect, *recvfrom*, *recvmsg*, *socket*

recvfrom

Receives data from a socket.

```
#include <pna.h>
long recvfrom(
   int s,                        /* socket descriptor */
   char *buf,                    /* packet buffer */
   int len,                      /* packet buffer length */
   int flags,                    /* packet attributes */
   struct sockaddr *from,        /* sender attributes */
   int *fromlen                  /* number of bytes recieved */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

```
-D_PNA_30_BACK
```

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

The `recvfrom()` system call is used to receive data from a socket. This system call is almost identical to `recv()`. The difference is `recvfrom()` may also return the address of the sender in the specified parameter.

Arguments

`s`	Specifies the socket from which data is received. The behavior of the system call depends on the socket type. Refer to `recv()` for more information.
`buf`	Points to the user buffer where data is stored.
`len`	Specifies the size of the buffer in bytes.
`flags`	Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in `<pna.h>`). Can also be set to 0.

MSG_OOB	Specifies that you want recvfrom() to read any out-of-band data present on the socket, rather than the regular in-band data.
MSG_PEEK	Specifies that you want recvfrom() to peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation sees the same data.
MSG_RAWMEM	Specifies that you have set buf to point to a linked list of mblks and len to the total size of the message.
MSG_INTERFACE	Specifies that you want the interface number of the NI on which the packet arrived to be stored in from.

from

If from is not a NULL pointer, recvfrom() fills in the sockaddr_in structure it points to with the address of the received data's sender.

The structure sockaddr_in is defined in <pna.h> and has the following format:

```
struct sockaddr_in {
   short sin_family;             /* must be AF_INET */
   unsigned short sin_port;      /* 16-bit port number */
   struct in_addr sin_addr;      /* 32-bit IP address */
   char sin_zero[8];             /* must be 0 */
   };
```

This structure cannot be packed.

If flags includes the MSG_INTERFACE constant, then the structure pointed to by from is filled with the structure sockaddr_intf. This is supported only for datagram sockets. This feature is useful with unnumbered links where it may not be clear which interface the packet arrived on.

The structure sockaddr_intf is defined in <pna.h> and has the following format:

```
struct sockaddr_intf {
   short sin_family;            /* must be AF_INET */
   unsigned short sin_port;     /* 16-bit port number */
   struct in_addr sin_addr;     /* 32-bit IP address */
   long sin_ifno;               /* 32-bit interface number */
   char sin_zero[4];            /* must be 0 */
   };
```

The field sin_ifno identifies the interface number of the incoming message's receiving interface.

`fromlen` An input-output parameter. On input, it should point to an integer equal to 16, which is the size in bytes of both `struct sockaddr_in` and `struct sockaddr_intf`.

Return Value

This system call returns either the number of bytes received or -1 if an error occurred. When the receive is shutdown by either end of the connection, a value of 0 is returned.

Error Codes

Refer to Appendix A.

See Also

recv, *recvmsg*, *socket*

recvmsg

Receives data from a socket.

```
#include <pna.h>
long recvmsg(
   int s,                /* socket descriptor */
   struct msghdr *msg,   /* packet */
   int flags             /* packet attributes */
   )
```

Description

The recvmsg() system call is used to receive data from a socket. This system call is similar to recvfrom() but requires fewer input parameters. Refer also to recv() for more details. Note that the MSG_RAWMEM option is not supported by this call.

Arguments

s — Specifies the socket from which data is received. The behavior of the system call depends on the socket type. Refer to recv() for more information.

msg — Points to the structure msghdr, which is defined in the file <pna.h> with the following format:

```
struct msghdr {
   char *msg_name;          /* optional address */
   int msg_namelen;         /* size of address */
   struct iovec *msg_iov;   /* scatter/gather array */
   int msg_iovlen;          /* # elements in msg_iov */
   char *msg_accrights;     /* access rights */
   int msg_accrightslen;    /* size of access rights buffer */
   };
```

This structure cannot be packed. The contents of the msghdr fields are described below.

msg_name — If the socket is unconnected, can specify the source from which it receives data.

msg_namelen — Specifies the length of the buffer pointed to by msg_name.

	`msg_iov`	Points to an array whose members (`msg_iov[0]`, ..,`msg_iov[msglen-1]`) specify the buffers in which the received data is stored. The `iovec` structure has the following format:
		`struct iovec {`
		`  char *iov_base:  /* base address */`
		`  int iov_len;  /* buffer length */`
		`};`
		This structure cannot be packed. Each `iovec` entry specifies the base address and length of an area in memory where data is stored. `recvmsg()` always fills an area completely before it goes to the next area.

	`msg_accrights`	Points to a buffer that receives the access rights information sent along with a message. This applies to messages that a UNIX host sends.
	`msg_accrightslen`	Specifies the length of the buffer pointed to by `msg_accrights`.
`flags`	Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in <pna.h>). It can also be set to 0.	
	`MSG_OOB`	Specifies that you want `recvmsg()` to read any out-of-band data present on the socket, rather than the regular in-band data.
	`MSG_PEEK`	Specifies that you want `recvmsg()` to peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation sees the same data.

Return Value

This system call returns the number of bytes received, or it returns -1 if an error occurs. When the receive is shutdown by either end of the connection, a value of 0 is returned.

Error Codes

Refer to Appendix A.

See Also

recv, *recvfrom*, *socket*

select

Checks the status of multiple sockets.

```
#include <pna.h>
long select(
   int width,                 /* largest descriptor list */
   fd_set *readset,           /* read descriptor list */
   fd_set *writeset,          /* write descriptor list
   fd_set *exceptset,         /* exception list */
   struct timeval *timeout    /* timeout for operation */
   )
```

Description

This system call is used to multiplex I/O requests among multiple sockets. Three sets of socket descriptors may be specified: a set of sockets from which to read, a set to which to write and a set that may have pending exceptional conditions.

Each set is actually a structure containing an array of long integer bit masks. The size of the array is set by the definition of FD_SETSIZE (in <pna.h>.) The array is long enough to hold one bit for each FD_SETSIZE socket descriptor.

If select() returns successfully, the three sets indicate which socket descriptors can be read, which can be written to, or which have exceptional conditions pending. A timeout value may be specified.

Arguments

width	Specifies the largest descriptor list given by readset, writeset, or exceptset.
readset	Points to a set of sockets from which to read.
writeset	Points to a set of sockets to which to write.
exceptset	Points to a set of sockets that may have an exceptional condition pending.

`timeout` Specifies a timeout option. If the fields in `timeout` are set to 0, `select()` returns immediately. If the `timeout` is a null pointer, `select()` blocks until a descriptor is selectable. The structure `timeval` is defined in the file `<pna.h>` and has the following format:

```
struct timeval {
   long tv_sec;     /* number of seconds */
   long tv_usec;   /* number of microsecons */
```

Usage

Macros

The status of a socket descriptor in a select mask can be tested with the `FD_ISSET(s, &mask)` macro, which returns a non-zero value if `s` is a member of the set mask, and 0 if it is not.

In addition, the macros `FD_SET(s, &mask)` and `FD_CLEAR(s, &mask)` are provided for adding and removing socket descriptors to and from a set. `s` is the socket descriptor, and `mask` points to a bit mask data structure. The macro `FD_ZERO(&mask)` is provided to clear the set and should be used before the set is used.

These macros are defined in the file `<pna.h>`.

Example

The following example shows how to use `select()` to determine if two sockets have available data:

```
fd_set read_mask;

struct timeval wait;

for (;;)
{
   wait.tv_sec = 1;      /* wait for 1 second */
   wait.tv_usec = 0;

   FD_ZERO (&read_mask);
   FD_SET (s1, &read_mask);
   FD_SET (s2, &read_mask);

   nb = select (FD_SETSIZE, &read_mask, (fd_set *) 0,
         (fd_set *) 0, &wait);
   if (nb <= 0)
```

```
        {
            /* error occurred or timed out */
        }

        if (FD_ISSET(s1, &read_mask))
        {
            /* socket 1 has data available */
        }

        if (FD_ISSET(s2, &read_mask))
        {
            /* socket 2 has data available */
        }
    }
```

If two tasks attempt to use `select()` on the same socket for the same conditions, an error occurs.

Return Value

A -1 is returned if an error occurs and the socket descriptor masks remain unchanged; a 0 is returned if a timeout occurs. On success a nonzero value is returned that indicates the number of descriptors on which events have occurred.

Error Codes

Refer to Appendix A.

See Also

accept, *connect*, *recv*, *send*

send

Sends data to a socket.

```
#include <pna.h>
long send(
   int s,         /* socket descriptor */
   char *buf,     /* packet */
   int len,       /* packet length */
   int flags      /* packet attributes */
   )
```

Description

The `send()` system call is used to send data to a foreign socket.

If no buffer space is available at the socket to hold the data to be transmitted, `send()` blocks the calling task unless the socket has been marked non-blocking.

Applications can use this call to pass messages to the pNA+ network manager in a linked list of *mblks* (message blocks) by setting the `MSG_RAWMEM` flag (see *Arguments*, below). Using mblks eliminates the data copy performed in the pNA+ network manager during the data transfer between the application and the pNA+ network manager.

Arguments

`s`	Specifies the local socket, which must be in a connected state. If `s` is a stream socket, the data is sent to the foreign socket that is connected to `s`. If `s` is a datagram socket, the data is sent to the socket that has been associated with `s` through a previous `connect()` system call. If `s` is a raw socket, the raw datagram is sent to the raw socket that has been associated with `s` through a previous `connect()` system call.
`buf`	Points to a buffer containing the data to send. If `s` is a datagram socket, the data that `buf` points to is a datagram.
`len`	Specifies the number of bytes in `buf`.
`flags`	Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in <pna.h>). It can also be set to 0.

MSG_OOB	Specifies that you want send() to send out-of-band data, rather than the regular in-band data.
MSG_DONTROUTE	Specifies that you want send() to turn on the socket flag SO_DONTROUTE for the duration of the send operation. The SO_DONTROUTE flag prohibits routing of outgoing data from the socket. Packets directed at unconnected nodes are dropped.
MSG_RAWMEM	Specifies that you have set buf to point to a linked list of mblks and len to the total size of the message.

Return Value

This system call returns the number of bytes sent or -1 if an error occurs.

Error Codes

Refer to Appendix A.

See Also

sendto, *sendmsg*, *socket*

sendmsg

Sends data to a socket.

```
#include <pna.h>
long sendmsg(
   int s,                /* socket descriptor */
   struct msghdr *msg,   /* packet structure */
   int flags             /* packet attributes */
   )
```

Description

The sendmsg() system call is used to send data to a foreign socket. This system call is similar to sendto(), but it requires fewer input parameters and uses the structure msghdr. For a complete description of this system call, refer to the sendto call description on page 6-64.

Note that the MSG_RAWMEM option is not supported by this call.

Arguments

s	Specifies the local datagram socket.
msg	Points to a msghdr structure. The msghdr structure is described in the recvmsg call description on page 6-54.
flags	Specifies usage options and is formed by ORing one or more of the following symbolic constants (defined in <pna.h>). It can also be set to 0.

MSG_OOB	Specifies that you want sendmsg() to send out-of-band data, rather than the regular in-band data.
MSG_DONTROUTE	Specifies that you want sendmsg() to turn on the socket flag SO_DONTROUTE for the duration of the operation. The SO_DONTROUTE flag prohibits routing of outgoing data from the socket. Packets directed at unconnected nodes are dropped.

Return Value

This system call returns the number of bytes sent or -1 if an error occurred.

Error Codes

Refer to Appendix A.

See Also

send, *sendto*, *socket*

sendto

Sends data to a socket.

```
#include <pna.h>
long sendto(
   int s,                    /* socket descriptor */
   char *buf,                /* packet */
   int len,                  /* packet length */
   int flags,                /* packet attribute */
   struct sockaddr *to,      /* destination socket type */
   int tolen                 /* size of sockaddr_in */
   )
```

NOTE: In the above structure, `sockaddr_in` can be used in place of `sockaddr` if the following compile option is specified on the command line (the -D option is valid for most compilers. Consult your compiler manual.):

```
-D_PNA_30_BACK
```

The structure for `sockaddr` is located in the `include/sys` directory in `sockcom.h`.

Description

The `sendto()` system call is used to send data to a foreign datagram socket. Although it is possible to use this system call with stream, datagram, or raw sockets, it is intended to be used only with datagram sockets or raw sockets.

If no buffer space is available at the socket to hold the datagram, `sendto()` blocks the calling task unless the socket has been marked non-blocking.

Applications can use this call to pass messages to the pNA+ network manager in a linked list of `mblks` (message blocks) by setting the `MSG_RAWMEM` flag. Using mblks eliminates the data copy performed in the pNA+ network manager during the data transfer between the application and the pNA+ network manager.

Arguments

`s`	Specifies the local socket.
`buf`	Points to a buffer that contains the data to send. The data pointed to by `buf` is called a datagram.

`len` Specifies the number of bytes in `buf`.

`flags` Specifies usage options and is the result of an OR operation performed on one or more of the following symbolic constants (defined in <pna.h>). It can also be set to 0.

`MSG_OOB` Specifies that you want `sendto()` to send out-of-band data, rather than the regular in-band data.

`MSG_DONTROUTE` Specifies that you want `sendto()` to turn on the socket flag `SO_DONTROUTE` for the duration of the send operation. The `SO_DONTROUTE` flag prohibits routing of outgoing data from the socket. Packets directed at unconnected nodes are dropped.

`MSG_RAWMEM` Specifies that you have set `buf` to point to a linked list of mblks and `len` to the total size of the message.

`MSG_INTERFACE` Specifies the outgoing interface of the message. If no route is found to the destination, then the interface number is specified in the argument `to`, which is a pointer to the structure `sockaddr_intf`. This is supported only for datagram or raw sockets. This is helpful with unnumbered links where there may not be a route to the destination.

`to` Specifies the destination socket address and is a pointer to either the `sockaddr_in` structure or the `sockaddr_intf` structure. `sockaddr_intf` is used *only* if the MSG_INTERFACE option is set in `flags`. These structures are defined in `<pna.h>` and have the following format:

```
struct sockaddr_in {
   short sin_family;           /* must be AF_INET */
   unsigned short sin_port;    /* 16-bit port number */
   struct in_addr sin_addr;    /* 32-bit IP address */
   char sin_zero[8];           /* must be 0 */
   };

struct sockaddr_intf {
   short sin_family;           /* must be AF_INET */
   unsigned short sin_port;    /* 16-bit port number */
   struct in_addr sin_addr;    /* 32-bit IP address */
   long sin_ifno;              /* 32-bit interface number */
   char sin_zero[4];           /* must be 0 */
   };
```

The field `sin_ifno` identifies the interface number of the outgoing packet's output interface.

The above structures cannot be packed.

`tolen` Specifies the size in bytes of either `struct sockaddr_in` or `struct sockaddr_intf` and must be 16.

Return Value

This system call returns the number of bytes sent and -1 if an error occurs.

Error Codes

Refer to Appendix A.

See Also

send, *sendmsg*, *select*, *socket*

set_id

Sets a task's user ID and group ID.

```
#include <pna.h>
long set_id(
   long userid,   /* user identity */
   long groupid,  /* group identity */
   long *groups   /* must be zero */
   )
```

Description

This system call sets the user ID and group ID of the calling task. These IDs are used for accessing NFS servers. Default values for the IDs are defined in the pNA+ Configuration Table.

Arguments

`userid`	Specifies the calling task's user ID.
`groupid`	Specifies the calling task's group ID.
`0`	Zero must be passed as a third argument (it is currently ignored).

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

None.

See Also

get_id

setsockopt Sets options on a socket.

```
#include <pna.h>
long setsockopt(
   int s,       /* socket descriptor */
   int level,   /* SOL_SOCKET, IPPROTO_TCP */
                /* or IPPROTO_IP */
   int optname, /* retrieval access */
   char *optval,/* modification data */
   int optlen   /* sizeof modification data */
   )
```

Description

The `setsockopt()` system call sets options associated with the specified socket. Socket level, IP protocol level, or TCP protocol level options can be set.

Arguments

`s`	Specifies the socket whose options are to be set.
`level`	Specifies the level of the option to be set. Must be `SOL_SOCKET` for socket level operations, `IPPROTO_TCP` for TCP protocol level operations, or `IPPROTO_IP` for IP protocol level operations. These options are defined in `<pna.h>`.
`optname`	Specifies the option to be set and uses a symbolic constant defined in `<pna.h>`. The symbolic constants available for each level are described below.
`optval`	Points to a buffer in which the option's value is specified. Most options are 32-bit values. A nonzero value means the option should be set, and a 0 means the option should be turned off.
`optlen`	Specifies the size of the value pointed to by `optval`.

Socket Level Options

`level` must be set to `SOL_SOCKET` for socket level operations and the `optname` value can be one of the following (defined in `<pna.h>`):

`SO_BROADCAST`	Allows broadcast datagrams on a socket.

SO_DONTROUTE — Indicates that the outgoing data should not be routed. Packets directed at unconnected nodes are dropped.

SO_KEEPALIVE — Maintains a connection by periodically transmitting a packet over socket s.

SO_LINGER — Controls the action taken when unsent messages are queued on a socket and a close() is executed. If the socket is a stream socket and SO_LINGER is set (l_onoff set to 1), the calling task blocks until it can transmit the data or until a timeout period (linger time, in seconds) expires. If SO_LINGER is disabled (l_onoff set to 0), the socket is deleted immediately. SO_LINGER uses the linger structure, which is defined in <pna.h> as follows:

```
struct linger {
   int l_onoff;    /* on/off option */
   int l_linger;   /* linger time in seconds */
   }
```

This structure cannot be packed.

SO_OOBINLINE — Requests that out-of-band data be placed in the normal data input queue as it is received; it becomes accessible through recv() calls without the MSG_OOB flag.

SO_RCVBUF — Adjusts the normal allocated input buffer size. The buffer size can be increased for high-volume connections or decreased to limit the possible backlog of data. The pNA+ network manager limits this value to 32 Kbytes.

NOTE: A UDP socket cannot receive a datagram that is equal in size to the size of the input (receive) buffer. This is due to the fact that the sbappendaddr() routine checks for space available to receive the datagram as:

```
size_of_datagram + size_of_sockaddr_structure
```

The size of the buffer must be increased by the size of the sockaddr structure to receive the datagram.

SO_REUSEADDR — Indicates that local addresses can be reused in a bind() call.

SO_REUSEPORT — Indicates that local addresses can be reused in a bind() call. For more information, see the Network Programming chapter in *pSOSystem System Concepts*.

SO_SNDBUF	Adjusts the normal allocated output buffer size. The buffer size can be increased for high-volume connections or decreased to limit the possible backlog of data. The pNA+ network manager limits this value to 32 Kbytes.

TCP Level Option

level must be set to IPPROTO_TCP for TCP protocol level operations. The argument optname can have the following value (defined in <pna.h>):

TCP_KEEPALIVE_CNT	Number of Keepalive strobes. Upon expiry of the Keepalive idle timer TCP will send a number of strobes separated by a fixed interval. If the other end fails to respond to the strobes (special TCP segments) then the TCP connection will be terminated. The default number of strobes in pNA+ are 8. Only valid if the SO_KEEPALIVE option is set above.
TCP_KEEPALIVE_IDLE	Keepalive idle time in TCP. If the connection has been idle for this time, the timer will expire causing TCP to send a special segment forcing the other end to respond. On demand-dial links for example the timer may be set long enough so as not to cause unnecessary traffic. The default in pNA+ is 2 hrs. The timer is in seconds. Only valid if the SO_KEEPALIVE option is set above.
TCP_KEEPALIVE_INTVL	Keepalive strobe interval. The strobes sent out by TCP upon expiration of the Keepalive idle timer are separated by a fixed interval. The default interval between the strobes is set to 75 seconds in pNA+. The timer is in seconds. Only valid if the SO_KEEPALIVE option is set above.
TCP_MSL	Maximum Segment Lifetime in TCP. This controls the TIME_WAIT or 2MSL timer in TCP which is set to twice the value of the MSL. The timer is used to validate connection termination and transmits remaining data in the send queue. It is a safeguard against sequence numbers being overlapped. If set to a low value it allows the sockets to be quickly deleted. The default in pNA+ is 30 seconds. The timer is in seconds.
TCP_NODELAY	Disables delay acknowledgment algorithm. Data is sent immediately over the network instead of waiting for the window to be full.

IP Level Options

`level` must be set to `IPPROTO_IP for IP protocol level operations`. The argument `optname` can have one of the following values (defined in <`pna.h`>):

`IP_ADD_MEMBERSHIP`

Join a multicast group. For this option, `optval` is a pointer to the `ip_mreq` structure (defined in <`pna.h`>). The group multicast address must be specified in the field `imr_mcastaddr`. This is a `CLASS-D` IP multicast address. The `imr_interface` parameter may be set to the IP address of a specific interface for which group membership will be enabled. The interface must of course support multicasting. Optionally the `imr_interface` parameter may be set to `INADDR_ANY` in which case pNA+ will decide the interface to join on (for example, if a route exists for a matching multicast address that specifies the interface). The maximum number of groups that can be joined per multicast socket is defined by the constant `IP_MAX_MEMBERSHIPS` in `pna.h`.

The structure below is defined in <`pna.h`> and used with the `IP_ADD_MEMBERSHIP` option.

```
struct ip_mreq {
   struct in_addr imr_mcastaddr;
      /* IP multicast address of group */
   struct in_addr imr_interface;
      /* local IP address of interface */
   };
```

IP_ADD_MEMBERSHIP_INTF

Similar to IP_ADD_MEMBERSHIP above. The optval is a pointer to the structure ip_mreq_intf (defined in <pna.h>). The only difference is that the interface is defined using the interface number. If the interface number is specified as -1 then pNA+ will select the interface based upon the routing table. This option is useful for unnumbered links because the IP address of the interface is not enough to identify the interface.

The structure below is defined in <pna.h> and used with the IP_ADD_MEMBERSHIP_INTF option.

```
struct ip_mreq_intf {
   struct in_addr imrif_mcastaddr;
      /* IP multicast address of group */
   long imrif_ifno;
      /* local interface number */
   };
```

IP_DROP_MEMBERSHIP

Leave a multicast group. optval is a pointer to the ip_mreq structure (defined in <pna.h>). The group multicast address to be dropped must be specified in imr_mcastaddr. The interface address could be set to INADDR_ANY unless it specifies the particular interface for which the group membership must be dropped.

The structure below is defined in <pna.h> and used with the IP_DROP_MEMBERSHIP option.

```
struct ip_mreq {
   struct in_addr imr_mcastaddr;
      /* IP multicast address of group */
   struct in_addr imr_interface;
      /* local IP address of interface */
   };
```

IP_DROP_MEMBERSHIP_INTF

Similar to IP_DROP_MEMBERSHIP above. The optval is a pointer to the structure ip_mreq_intf (defined in <pna.h>). The only difference is that the interface is defined using the interface number. If the interface number is specified as -1 then pNA+ will select the interface based upon the routing table. This option is useful for unnumbered links because the IP address of the interface is not enough to identify the interface.

The structure below is defined in <pna.h> and used with the IP_DROP_MEMBERSHIP_INTF option.

```
struct ip_mreq_intf {
   struct in_addr imrif_mcastaddr;
      /* IP multicast address of group */
   long imrif_ifno;
      /* local interface number */
   };
```

IP_HDRINCL

Specifies that the IP header will be included in the output packets. The following fields will be set by pNA+ if they are set to 0 in the included IP header: IP identification number and IP source address. The fragmentation offset and checksum fields are always computed by pNA+. The rest of the IP header fields must be set appropriately.

IP_MULTICAST_IF

Specifies the outgoing interface for multicast packets. optval is a pointer to struct in_addr. If set to INADDR_ANY then the routing table will be used to select an appropriate interface.

IP_MULTICAST_INTF

Specifies the outgoing interface for multicast packets. The outgoing interface is defined by its interface number. The optval is a pointer to a long. If set to -1 then the routing table will be used to select an appropriate interface. This option is useful for unnumbered links because the IP address of the interface is not enough to identify the interface.

IP_MULTICAST_LOOP	Specifies whether or not to loopback multicast packets. optval is a pointer to an unsigned char. By default the packets are looped back (IP_DEFAULT_MULTICAST_LOOP). A value of 0 disables loopback.
IP_MULTICAST_TTL	Specifies the time-to-live for outgoing IP multicast datagrams. optval is a pointer to an unsigned char. The default is IP_DEFAULT_MULTICAST_TTL.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

getsockopt, *socket*

shr_socket

Obtains a new socket descriptor for an existing socket.

```
#include <pna.h>
int shr_socket(
   int s,    /* socket descriptor */
   int tid   /* task identity */
   )
```

Description

This system call is used to obtain a new socket descriptor for an existing socket. The new socket descriptor can be used by the task with the specified ID to reference the socket in question.

This system call is provided for applications that implement UNIX-style server programs, which normally incorporate the UNIX `fork()` call.

Arguments

`s`	Specifies the existing socket descriptor to be shared.
`tid`	Specifies the task ID of a task that seeks to access the same socket.

Return Value

This system call returns a socket descriptor if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

shutdown

Terminates all or part of a full-duplex connection.

```
#include <pna.h>
long shutdown(
   int s,    /* socket descriptor */
   int how   /* shutdown mechanism */
   )
```

Description

This system call is used to terminate all or part of a full-duplex connection on a specified socket. The socket may be shut down for sending, receiving, or both.

Arguments

s	Specifies the socket to be shut down.
how	Specifies the shutdown mechanism. Three options are available, as follows:
	0 — No further receives are allowed on the socket.
	1 — No further sends are allowed on the socket.
	2 — No further sends or receives are allowed on the socket.

Return Value

This system call returns 0 if successful; otherwise, it returns -1.

Error Codes

Refer to Appendix A.

See Also

connect, *socket*

socket

Creates a socket.

```
#include <pna.h>
int socket(
   int domain,    /* socket domain */
   int type,      /* socket type carrier */
   int protocol   /* socket protocol class */
   )
```

Description

The `socket()` system call creates a new socket and returns its socket descriptor. The socket is an endpoint of communication.

Arguments

`domain` Specifies the socket domain and must be set to `AF_INET`.

`type` Specifies one of the following types of sockets (defined in the `<pna.h>` file):

- `SOCK_STREAM` Defines a stream socket, which uses TCP to provide a reliable connection-based communication service.
- `SOCK_DGRAM` Defines a datagram socket, which uses UDP to provide a datagram service.
- `SOCK_RAW` Defines a raw socket, which uses the protocol specified by `protocol` for a raw datagram service.

`protocol` Specifies the network protocol and can be 0, TCP, UDP, or any other protocol. For raw sockets the protocol can have any value except TCP or UDP. A protocol number of zero acts as a wildcard for raw sockets, accepting any raw IP packet.

Return Value

This system call returns a socket descriptor, or a -1 if an error occurs.

Error Codes

Refer to Appendix A.

See Also

accept, *bind*, *close*, *connect*, *listen*, *setsockopt*, *getsockopt*

7 pRPC+ System Calls

This chapter provides information on the system calls in the pRPC+ component of pSOSystem. Each call's section includes its syntax, a description, its arguments, its return value, and any error codes that it can return. Where applicable, the section also includes the headings "Notes" and "See Also." "Notes" provides any important information not specifically related to the call's description, and "See Also" indicates other calls that have related information.

If you need to look up a system call by its functionality, refer to Table 7-1 on page 7-2, which lists the pRPC+ calls alphabetically by component and provides a brief description of each call. *pRPC+ System Calls* *on page 7-2* shows all of the services supported by the pRPC+ library.

For more information on error codes, refer to Appendix A, *Error Codes*, which lists the codes numerically and gives the pSOSystem calls that are associated with each one.

`pmap_start()` need not be called before making any pRPC+ call. This is done automatically on the first call to pRPC+. TCP/IP for OpEN stack should be up before making any pRPC for OpEN call.

pRPC+ version 2.0 is backward compatible with the old release of pRPC+. The new version lets users configure pRPC+ to be a sub-component of pSE and to use OpEN sockets instead of pNA sockets. With this version of pRPC+, both NFS server and client can be run over TCP/IP for OpEN.

TABLE 7-1 pRPC+ System Calls

Name	Description	Page
clnt_control	Change the internal client creation timeout values.	7-5
get_fdset	Returns the bit mask that corresponds to readable RPC sockets.	7-7
rpc_getcreateerr	Returns the reason for an RPC client handle creation failure.	7-8

pRPC+ System Calls

The following list shows all of the services supported by the pRPC+ library. All routines implement standard ONC RPC/XDR services, except for those marked with an asterisk. The calls marked with an asterisk (*) are described in detail in this chapter (see also Table 7-1).

auth_destroy	authnone_create	authunix_create
authunix_create_default	callrpc	clnt_broadcast
clnt_call	clnt_create	clnt_destroy
clnt_freeres	clnt_geterr	clnt_perrno
clnt_perror	clnt_sperrno	clnt_sperror
*clnt_control**	clnt_pcreateerror	clntraw_create
clnt_spcreateerror	clnt_udp_bufcreate	clntudp_create
clnttcp_create	*get_fdset**	pmap_getmaps
pmap_getport	pmap_rmtcall	pmap_set
pmap_unset	registerrpc	*rpc_getcreateerr**
svcerr_auth	svcerr_decode	svcerr_noproc
svcerr_noprog	svcerr_progvers	svcerr_systemerr
svcerr_weakauth	svcfd_create	svcraw_create
svctcp_create	svcudp_create	svc_destroy
svc_fdset	svc_freeargs	svc_getargs
svc_getcaller	svc_getreq	svc_getreqset
svc_run	svc_sendreply	svc_register

svc_unregister	xdrmem_create	xdrrec_create
xdrrec_endofrecord	xdrrec_eof	xdrrec_readbytes
xdrrec_skiprecord	xdrstdio_create	xdr_accepted_reply
xdr_array	xdr_authunix_parms	xdr_bool
xdr_bytes	xdr_callhdr	xdr_callmsg
xdr_char	xdr_destroy	xdr_double
xdr_enum	xdr_float	xdr_free
xdr_getpos	xdr_inline	xdr_int
xdr_long	xdr_opaque	xdr_opaque_auth
xdr_pmap	xdr_pmaplist	xdr_pointer
xdr_reference	xdr_rejected_reply	xdr_replymsg
xdr_setpos	xdr_short	xdr_string
xdr_union	xdr_u_char	xdr_u_int
xdr_u_long	xdr_u_short	xdr_void
xdr_vector	xdr_wrapstring	xprt_register
xprt_unregister		

clnt_control

```
#include <rpc.h>
bool_t clnt_control(
          CLIENT *clnt,    /* Client handle
          u_int req,       /* request type CLSET_TIMEOUT/
                            * CLSET_RETRY_TIMEOUT */
          char *info       /* value */
   )
```

Description

This is an extension to the `clnt_control` call. This extension lets you change the internal client creation timeout values. During a `clnt_create()` call, pRPC attempts to contact the portmapper on the machine on which the server is running. The pRPC waits for the default timeout value (60 seconds) for a response from the portmapper task. This timeout may be too long for some applications. In such cases, you will be able to change the timeout values.

The `clnt_control` call with a NULL client handle value lets you change one of the timeout values. The req value 0 (`CLSET_TIMEOUT`) will change the total timeout and 1 (`CLSET_RETRY_TIMEOUT`) will change the retry timeout.

The new timeout value will be used by all subsequent calls.

Arguments

`clnt`	Client handle.
`req`	Request type (0=`CLSET_TIMEOUT`;1=`CLSET_RETRY_TIMEOUT`).
`info`	Pointer to timeval structure. The timeval structure is:

```
struct timeval {
  long tv_sec;    /* # of seconds */
  long tv_usec;   /* # of microseconds */
}:
```

Return Value

`TRUE(1)` on success and `FALSE(0)` upon failure.

Error Codes

None.

See Also

Please refer to the `clnt_control` manual page for normal behavior of this call.

get_fdset

```
#include <rpc.h>
void get_fdset(
   fd_set *read_mask    /* server's read file
                         * descriptor bit mask */
   )
```

Description

This pRPC+ service call provides access to the task-specific equivalent of the ONC RPC `svc_fdset` global variable.

Arguments

`read_mask`	Points to the location where `get_fdset()` copies the contents of the `svc_fdset` variable. Memory pointed to by `read_mask` should be preallocated. The returned `read_mask` can serve as the read descriptor list argument to the pNA+ `select()` service call. The value that `select()` returns in place of the input read descriptor list can serve as an input argument to the pRPC+ `svc_getreqset()` call.

Return Value

None.

Error Codes

None.

See Also

The `svc_getreqset` description in other ONC RPC documentation.

rpc_getcreateerr

```
#include <rpc.h>
void rpc_getcreateerr(
   struct rpc_createerr *err      /* error buffer */
   )
```

Description

This pRPC+ service call provides access to the task-specific equivalent of the ONC RPC `rpc_createerr` global variable.

Arguments

`rpc_createerr`	Points to the location where `rpc_getcreateerr()` copies the contents of the `rpc_createerr` variable. The creation routines for the RPC client handle use `rpc_createerr` to store the reason for a creation failure. Application programs do not require access to this variable: the standard routines `clnt_pcreateerror()` and `clnt_spcreateerr()` return a textual description of the failure.

Return Value

None.

Error Codes

None.

See Also

The `clnt_pcreateerror` and `clnt_spcreateerror` descriptions in other ONC RPC documentation.

8 pROBE+ and ESp System Calls

This chapter provides detailed descriptions of the system calls supported by pROBE+ and ESp. The calls are listed alphabetically, with a multipage section of information for each call. Each call's section includes its syntax, a detailed description, its arguments, and its return value.

The ESp cross-system visual analyzer graphically displays your embedded application's activity on a host terminal and allows you to analyze the application's performance. ESp accepts one system call, `log_event()`, through its target-resident application monitor, pMONT.

pROBE+ is pSOSystem's target debugger and analyzer. pROBE+ accepts two system calls: `db_input()` and `db_output()`.

If you need to look up a system call by its functionality, refer to Table 8-1, which lists the calls alphabetically by component and provides a brief description of each call.

For more information on error codes, refer to Appendix A, *Error Codes*, which lists the codes numerically and gives the pSOSystem calls associated with each one.

TABLE 8-1 pROBE+ and ESp System Calls

Name	Tool	Description	Page
`db_input`	pROBE+	Prompts and gets input from the high-level debugger.	8-3
`db_output`	pROBE+	Outputs a string to the high-level debugger.	8-5
`log_event`	ESp	Logs an event on ESp's target-resident application monitor, pMONT.	8-6

db_input

Prompts and gets input from the high-level debugger.

```
long db_input (
   char *inbuf,                /* user input buffer */
   unsigned long inbuf_len,    /* user input buffer length */
   char *prompt,               /* prompt string */
   unsigned long prompt_len,   /* prompt string length */
   unsigned long *nbytes       /* number of bytes written */
   )
```

Description

This system call passes the string `prompt` to the high-level debugger (HLD) to be printed on the output screen and waits for input back from the HLD. Upon receipt of the input, it copies up to `inbuf_len` bytes into `inbuf` and writes `inbuf`'s length into `nbytes`.

Arguments

`inbuf`	Points to the buffer in which `db_input()` writes input from the HLD.
`inbuf_len`	Specifies the number of bytes of input data to be written in `inbuf`.
`prompt`	Points to the string to be passed to the HLD as a prompt.
`prompt_len`	Specifies the number of characters in `prompt`.
`nbytes`	Points to the buffer in which `db_input()` writes the actual length of `inbuf`.

Return Value

This call returns 0 on success and an error code on failure.

Error Codes

Hex	Mnemonic	Description
Not set at this time.	None.	Lost connection to host debugger.

Notes

- Requires support from a host debugger.

See Also

db_output

db_output Outputs a string to the high-level debugger.

```
long db_output (
   char *str,              /* string */
   unsigned long length    /* string length */
   )
```

Description

This system call sends a character string to the high-level debugger (HLD) to be printed on its output screen.

Arguments

`str`	Points to the string to be output to the HLD.
`length`	Specifies the length of the string.

Return Value

This call returns 0 on success and an error code on failure.

Error Codes

Hex	Mnemonic	Description
Not set at this time.	None.	Lost connection to host debugger.

See Also

db_input

log_event

Logs an event on ESp's target-resident application monitor, pMONT.

```
unsigned long log_event (
   unsigned long user_event_id,/* user-defined event ID */
   unsigned long event_data    /* user-defined event data */
   )
```

Description

This call logs an event in pMONT's trace buffer. The `log_event()` call takes effect when the ESp data collection run begins.

Arguments

`user_event_id`	Specifies an ID number for the current call to `log_event()`. The maximum allowable ID number is 0xff. Providing an ID number for each call to `log_event()` helps you keep track of user-events.
`event_data`	An optional word or words you can use to store data associated with the event.

Return Value

This call always returns 0.

Error Codes

None.

Notes

pMONT+ uses pSOS+ objects for its functionality. Therefore, some user-defined entries in the pSOS+ configuration table affect pMONT+ behavior and can even increase the likelihood of error messages. For example, an insufficient number of message buffers may result in a sudden break in the host/target connection.

pMONT+ uses the system-wide buffer pool to post messages to its queues. So, you need to consider this when specifying `kc_nmsgbuf` in the pSOS+ configuration ta-

ble. How much pMONT+ affects the specification of `kc_nmsgbuf` depends on the monitoring demands that you expect pMONT+ to meet.

The rate at which the system-wide buffer pool is replenished depends on the communication medium used to communicate with the host. With a network, for example, pMONT+ replenishes the buffer relatively quickly. You can estimate the requirements for the application by considering the following ESp behaviors:

- A message is posted to a queue every time an object is created or deleted.
- A message is posted to a queue every time pMONT+ receives a request.
- A message is posted when data collection ends under any of the buffer management options.
- Under the Transmit Buffer Management option, pMONT+ periodically sends data to the host. Every time it does so, pMONT+ posts a message to a queue. The rate at which this occurs depends on the data collection configuration and the application.
- A message is posted to a queue at the end of every user-specified Perfmeter update period.
- If the Stack Checking feature is on, a message is posted every time a stack warning is generated.

A Error Codes

This appendix is a collection of tables of pSOSystem error codes, intended to help you identify which system call returned a specific error code. Each table lists the codes belonging to a single pSOSystem component (i.e., pSOS+, pHILE+, etc.) The table entry for each code includes a hexadecimal number, a brief description (including the error mnemonic), and a list of the system calls that can return the error.

pSOSystem components return error codes in the following ways:

- pSOS+ and pHILE+ return error codes as function return values.
- pREPC+, pLM+, pNA+, pRPC+, and pROBE+ load the error code into an internal variable that can be read through the macro `errno()`. If the return value of a pREPC+, pLM+, pNA+, pRPC+, or pROBE+ system call indicates an error, your application should examine the `errno` variable to determine the cause of the error. See the description of `errno()` on page 4-44 for more information.
- pMONT+ stores the error code so that, upon the next established connection, the ESp log window displays the error code as `prevErr`.

Table A-1 lists the error code ranges of pSOSystem components, libraries, and drivers. Error code values are in hexadecimal notation, with a space inserted after every byte for readability.

TABLE A-1 Error Code Origins

Error Code Range		Origin	Defined in	Refer to page
From	To			
00 00 00 01	00 00 0F FF	pSOS+, pSOS+m	psos.h	A-4
00 00 10 00	00 00 1F FF	pROBE_init	errno.h	A-19
00 00 20 00	00 00 2F FF	pHILE+	phile.h	A-20
00 00 30 00	00 00 3F FF	pREPC+	errno.h	A-45
00 00 40 00	00 00 4F FF	pLM+	errno.h	A-46
00 00 50 00	00 00 5F FF	pNA+, pRPC+, pX11+	pna.h	A-49
00 00 60 00	00 00 6F FF	(reserved)		
00 01 00 00	00 FF FF FF	(reserved)		
01 10 00 00	01 1F FF FF	Networking libraries		
01 20 00 00	01 20 00 FF	MMUlib	mmulib.h	
01 20 01 00	01 20 01 FF	Loader		
01 20 02 00	00 FF FF FF	(reserved for pSOSystem libraries)		
10 00 00 00	10 00 00 FF	NI_SMEM driver	drv_intf.h	A-58
10 00 01 00	10 00 01 FF	KI_SMEM driver	drv_intf.h	A-59
10 00 02 00	10 00 FF FF	(reserved for pSOSystem)		
10 01 00 00	10 01 FF FF	serial driver	drv_intf.h	A-59
10 02 00 00	10 02 FF FF	tick timer driver	drv_intf.h	A-61
10 03 00 00	10 03 FF FF	(reserved for pSOSystem)		
10 04 00 00	10 04 FF FF	RAM disk driver	drv_intf.h	A-61

TABLE A-1 Error Code Origins (Continued)

Error Code Range		Origin	Defined in	Refer to page
From	To			
10 05 00 00	10 05 FF FF	(reserved for pSOSystem)		
10 06 00 00	10 06 FF FF	TFTP driver	`drv_intf.h`	A-61
10 07 00 00	10 07 FF FF	SLIP driver		
10 08 00 00	10 08 FF FF	(reserved for pSOSystem)		
10 09 00 00	10 09 FF FF	IDE driver		A-62
10 0A 00 00	10 0A FF FF	FLOPPY (FLP) driver		A-63
10 0B 00 00	10 4F FF FF	(reserved for pSOSystem)		
10 0C 00 00	10 0C FF FF	HTTP driver		A-64
10 0D 00 00	10 0D FF FF	PIPE driver		A-64
10 0E 00 00	10 0E FF FF	RDIO driver		A-65
10 0F 00 00	10 3F FF FF	(reserved for pSOSystem)		
10 40 00 00	10 4F FF FF	LAN driver	`drv_intf.h`	A-65
10 50 00 00	10 5F FF FF	SCSI driver	`drv_intf.h`	A-66
10 60 00 00	10 6F FF FF	Parallel Port Driver		A-68
10 70 00 00	10 7F FF FF	MAPIO driver		
10 80 00 00	1F FF FF FF	(reserved for pSOSystem)		
20 00 00 00	FF FF FF FF	(reserved for application use) Error codes in the range 20 00 00 00 - FF FF FF FF are reserved for user-added drivers. Any custom drivers you add to pSOSystem should return error codes in this range. Other ranges marked (reserved) are reserved for use by future additions to pSOSystem and should not be used by custom drivers.		

A

A.1 pSOS+ Error Codes

All pSOS+ error codes are returned as function return values (rather than an `errno` variable); they have a value between 0 and 0xfff.

Table A-2 lists all error codes returned by the pSOS+ component. Each listing includes the error code's hexadecimal number, it's mnemonic and description, and the pSOS+ system calls that can return it. The error code mnemonics are also defined in `<psos.h>`.

The term *`object`* represents the applicable service group type (task, partition, queue, semaphore, and so on).

TABLE A-2 pSOS+ Error Codes

Hex	Mnemonic and Description	System Call(s)
0x01	`ERR_TIMEOUT`: Timed out; returned only if a timeout was requested. For co_unregister, the `CO_WAIT` flag must have also been specified. For ev_receive, the `EV_WAIT` flag must have also been specified.	`co_unregister, cv_wait, ev_receive, mu_lock, q_receive, q_vreceive, rn_getseg. sm_p`
0x03	`ERR_SSFN`: Illegal system service function number.	`sm_av`
0x04	`ERR_NODENO`: Node specifier out of range.	`cv_ident, k_terminate, mu_ident, pt_ident, q_ident, q_vident, sm_ident, t_ident`

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x05	ERR_OBJDEL: *object* has been deleted.	as_send, cv_abroadcast, cv_asignal, cv_broadcast, cv_delete, cv_info, cv_signal, cv_wait, ev_asend, ev_send, mu_delete, mu_info, mu_lock, mu_setceil, mu_unlock, pt_delete, pt_getbuf, pt_info, pt_retbuf, pt_sgetbuf, q_asend, q_aurgent, q_avsend, q_avurgent, q_broadcast, q_delete, q_info, q_notify, q_receive, q_send, q_urgent, q_vbroadcast, q_vdelete, q_vinfo, q_vnotify, q_vreceive, q_vsend, q_vurgent, rn_delete, rn_getseg, rn_info, rn_retseg, sm_av, sm_delete, sm_info, sm_notify, sm_p, sm_v, t_addvar, t_delete, t_delvar, t_getreg, t_info, t_restart, t_resume, t_setpri, t_setreg, t_start, t_suspend, t_tslice, tsd_delete, tsd_getval, tsd_ident, tsd_info, tsd_setval

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x06	ERR_OBJID: *object_id* is incorrect; failed validity check.	as_send, cv_abroadcast, cv_asignal, cv_broadcast, cv_delete, cv_info, cv_signal, cv_wait, ev_asend, ev_send, mu_delete, mu_info, mu_lock, mu_setceil, mu_unlock, pt_delete, pt_info, pt_getbuf, pt_retbuf, pt_sgetbuf, q_asend, q_aurgent, q_avsend, q_avurgent, q_broadcast, q_delete, q_info, q_notify, q_receive, q_send, q_urgent, q_vbroadcast, q_vdelete, q_vinfo, q_vnotify, q_vreceive, q_vsend, q_vurgent, rn_delete, rn_getseg, rn_info rn_retseg, sm_av, sm_delete, sm_info, sm_notify, sm_p, sm_v, t_addvar, t_delete, t_delvar, t_getreg, t_info, t_restart, t_resume, t_setpri, t_setreg, t_start, t_suspend, t_tslice, tsd_delete, tsd_getval, tsd_ident, tsd_info, tsd_setval

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x07	ERR_OBJTYPE: *object* type doesn't match *object* ID; failed validity check.	as_send, cv_abroadcast, cv_asignal, cv_broadcast, cv_delete, cv_info, cv_signal, cv_wait, ev_asend, ev_send, mu_delete, mu_info, mu_lock, mu_setceil, mu_unlock, ob_roster, pt_delete, pt_getbuf, pt_info, pt_retbuf, pt_sgetbuf, q_asend, q_aurgent, q_avsend, q_avurgent, q_broadcast, q_delete, q_info, q_notify, q_receive, q_send, q_urgent, q_vbroadcast, q_vdelete, q_vinfo, q_vnotify, q_vreceive, q_vsend, q_vurgent, rn_delete, rn_getseg, rn_info, rn_retseg, sm_av, sm_delete, sm_info, sm_notify, sm_p, sm_v, t_addvar, t_delete, t_delvar, t_getreg, t_info, t_restart, t_resume, t_setpri, t_setreg, t_start, t_suspend, t_tslice, tsd_delete, tsd_getval, tsd_info, tsd_setval

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x08	ERR_OBJTFULL: Node's *object* table full.	cv_create, mu_create, pt_create, q_create, q_vcreate, rn_create, sm_create, t_create, tsd_create
0x09	ERR_OBJNF: Named *object* not found.	cv_ident, mu_ident, pt_ident, q_ident, q_vident, rn_ident, sm_ident, t_ident, tsd_ident
0x0D	ERR_RSTFS: Informative; files may be corrupted on restart.	t_restart
0x0E	ERR_NOTCB: Exceeds node's maximum number of tasks. The maximum number of tasks that can be simultaneously active is defined by the kc_ntask entry in the pSOS+ Configuration Table.	t_create
0x0F	ERR_NOSTK: Insufficient space in Region 0 to create stack.	t_create
0x10	ERR_TINYSTK: Stack too small.	t_create
0x11	ERR_PRIOR: Priority out of range. For mu, the MU_PRIO_PROTECT flag has been specified. The priority must be more than 0 and less than 256.	mu_create, mu_setceil, t_create
0x12	ERR_ACTIVE: Task already started.	t_start
0x13	ERR_NACTIVE: Cannot restart; this task never was started.	t_restart
0x14	ERR_SUSP: Task already suspended.	t_suspend
0x15	ERR_NOTSUSP: The task was not suspended.	t_resume
0x16	ERR_SETPRI: Cannot change priority; new priority out of range.	t_setpri
0x17	ERR_REGNUM: Register number out of range.	t_getreg, t_setreg

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x18	ERR_DELFS: pHILE+ resources in use.	t_delete
0x19	ERR_DELLC: pREPC+ resources in use.	t_delete
0x1A	ERR_DELNS: pNA+ resources in use.	t_delete
0x1B	ERR_RNADDR: Starting address not on long word boundary.	rn_create
0x1C	ERR_UNITSIZE: Illegal *unit_size* — unit size not power of 2 or less than 16 bytes.	rn_create
0x1D	ERR_TINYUNIT: *length* too large (for given *unit_size*.)	rn_create
0x1E	ERR_TINYRN: Cannot create; region length too small to hold RNCB.	rn_create
0x1F	ERR_SEGINUSE: Cannot delete; one or more segments still in use.	rn_delete
0x20	ERR_ZERO: Cannot getseg; request size of zero is illegal.	rn_getseg
0x21	ERR_TOOBIG: Cannot getseg; request size is too big for region.	rn_getseg
0x22	ERR_NOSEG: No free segment; only if RN_NOWAIT attribute used.	rn_getseg
0x23	ERR_NOTINRN: Segment does not belong to this region.	rn_retseg
0x24	ERR_SEGADDR: Incorrect segment starting address.	rn_retseg
0x25	ERR_SEGFREE: Segment is already unallocated.	rn_retseg
0x26	ERR_RNKILLD: Cannot getseg; region deleted while waiting.	rn_getseg
0x27	ERR_TATRNDEL: Informative only; there were tasks waiting.	rn_delete
0x28	ERR_PTADDR: Starting address not on long word boundary.	pt_create

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x29	ERR_BUFSIZE: Buffer size not power of 2, or less than 4 bytes.	pt_create
0x2A	ERR_TINYPT: Length too small to hold the partition control information (PTCB).	pt_create
0x2B	ERR_BUFINUSE: Cannot delete; one or more buffers still in use.	pt_delete
0x2C	ERR_NOBUF: Cannot allocate; partition out of free buffers.	pt_getbuf, pt_sgetbuf
0x2D	ERR_BUFADDR: Incorrect buffer starting address.	pt_retbuf
0x2F	ERR_BUFFREE: Buffer is already unallocated.	pt_retbuf
0x30	ERR_KISIZE: Global queue maxlen too large for KI.	q_vbroadcast, q_vcreate, q_vreceive, q_vsend, q_vurgent
0x31	ERR_MSGSIZ: Message too large.	q_vbroadcast, q_avsend, q_avurgent, q_vsend, q_vurgent
0x32	ERR_BUFSIZ: Buffer too small.	q_vreceive
0x33	ERR_NOQCB: Can't allocate QCB: exceeds node's active queue maximum.	q_create, q_vcreate
0x34	ERR_NOMGB: Cannot allocate private (message) buffers; too few available.	q_asend, q_aurgent, q_create, q_send, q_urgent, q_vcreate
0x35	ERR_QFULL: Message queue at length limit.	q_asend, q_aurgent, q_avsend, q_avurgent, q_send, q_urgent, q_vsend, q_vurgent
0x36	ERR_QKILLD: Queue deleted while task waiting.	q_receive, q_vreceive
0x37	ERR_NOMSG: Queue empty: this error returns only if Q_NOWAIT selected.	q_receive, q_vreceive
0x38	ERR_TATQDEL: Informative only: tasks were waiting at the queue.	q_delete, q_vdelete

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x39	ERR_MATQDEL: Information only: messages were pending in the queue.	q_delete, q_vdelete
0x3A	ERR_VARQ: Queue is variable length.	q_asend, q_aurgent, q_broadcast, q_delete, q_info, q_receive, q_send, q_urgent
0x3B	ERR_NOTVARQ: Queue is not variable length.	q_avsend, q_avurgent, q_vbroadcast, q_vdelete, q_vinfo, q_vreceive, q_vsend, q_vurgent
0x3C	ERR_NOEVS: Selected events not pending: this error code is returned only if the EV_NOWAIT attribute was selected.	ev_receive
0x3D	ERR_NOTIFY: Could not send a notification event. Error occurred while sending notification of message availability.	q_asend, q_aurgent, q_avsend, q_avurgent, q_send, q_urgent, q_vsend, q_vurgent, sm_av, sm_v
0x3E	ERR_NOTINASR: Illegal, not called from an ASR.	as_return
0x3F	ERR_NOASR: Task has no valid ASR.	as_send
0x41	ERR_NOSCB: Exceeds node's maximum number of semaphores.	sm_create
0x42	ERR_NOSEM: No semaphore: this error code returns only if SM_NOWAIT was selected.	sm_p
0x43	ERR_SKILLD: Semaphore deleted while task waiting.	sm_p
0x44	ERR_TATSDEL: Informative only; there were tasks waiting.	sm_delete
0x45	ERR_BOUND: Maximum number of tokens are already available for a semaphore that was created with the SM_BOUNDED flag.	sm_v

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x46	ERR_INVBOUND: ZERO bound count is invalid.	sm_create
0x47	ERR_NOTIME: System time and date not yet set.	tm_evafter, tm_evevery, tm_evwhen, tm_get, tm_wkwhen
0x48	ERR_ILLDATE: *date* input out of range.	tm_evwhen, tm_set, tm_wkwhen
0x49	ERR_ILLTIME: *time* input out of range.	tm_evwhen, tm_set, tm_wkwhen
0x4A	ERR_ILLTICKS: *ticks* input out of range.	tm_evwhen, tm_set, tm_wkwhen
0x4B	ERR_NOTIMERS: Exceeds maximum number of configured timers.	tm_evafter, tm_evevery, tm_evwhen
0x4C	ERR_BADTMID: tmid invalid.	tm_cancel
0x4D	ERR_TMNOTSET: Timer not armed or already expired.	tm_cancel
0x4E	ERR_TOOLATE: Too late; *date* and *time* input already in the past.	tm_evwhen, tm_wkwhen
0x53	ERR_ILLRSC: *object* not created from this node.	cv_delete, cv_info, mu_delete, mu_info, pt_delete, pt_info, q_delete, q_info, q_notify, q_vdelete, q_vinfo, q_vnotify, rn_info, sm_delete, sm_info, sm_notify, t_addvar, t_delete, t_delvar, t_info, t_restart, t_start, tsd_getval, tsd_info, tsd_setval
0x54	ERR_NOAGNT: Cannot wait; the remote node is out of Agents.	cv_wait, mu_lock, q_receive, q_vreceive, sm_p

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x65	ERR_STALEID: *object* does not exist any more.	cv_abroadcast, cv_asignal, cv_broadcast, cv_ident, cv_signal, cv_wait, ev_send, mu_ident, mu_lock, mu_setceil, mu_unlock, pt_getbuf, pt_retbuf, pt_sgetbuf, q_asend, q_aurgent, q_avsend, q_avurgent, q_broadcast, q_receive, q_send, q_urgent, q_vbroadcast, q_vreceive, q_vsend, q_vurgent, sm_p, sm_v, t_getreg, t_resume, t_setpri, t_setreg,
0x66	ERR_NDKLD: Remote node is no longer in service.	cv_broadcast, cv_ident, cv_signal, cv_wait, mu_ident, mu_lock, mu_unlock, q_receive, q_vreceive, sm_p
0x67	ERR_MASTER: Cannot terminate master node.	k_terminate
0x70	ERR_NOMUCB: Exceeds node's maximum number of mutexes.	mu_create
0x71	ERR_NOGLOBAL: The MU_GLOBAL flag cannot be specified with the MU_PRIO_INHERIT flag.	mu_create
0x72	ERR_TATMUDEL: Informative only: there were tasks waiting.	mu_delete
0x73	ERR_NOTOWNED: The mutex is not owned by the calling task.	cv_wait, mu_unlock

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x74	ERR_NORECURSIVE: The mutex is already locked by the calling task and it was created with MU_NORECURSIVE flag.	mu_lock
0x75	ERR_MULOCKED: Some other task has already locked the mutex; this error is returned only if MU_NOWAIT flag has been specified.	mu_lock
0x76	ERR_MUKILLED: Mutex deleted while the task was waiting on it.	cv_wait, mu_lock,
0x77	ERR_NOFIFO: MU_FIFO flag cannot be specified with MU_PRIO_INHERIT or MU_PRIO_PROTECT.	mu_create
0x79	ERR_OWNMUTX: Task was created with T_NORELMU flag set and currently owns some mutexes.	t_delete
0x7A	ERR_OUTCEIL: Task's current priority exceeds the ceiling priority of the guarding mutex.	cv_wait, mu_lock
0x80	ERR_NOCVCB. Exceeds node's maximum number of condition variables.	cv_create
0x81	ERR_TATCVDEL: Informative only: there were tasks waiting.	cv_delete
0x82	ERR_CVKILLD: Condition variable deleted while the task was waiting on it.	cv_wait
0x83	ERR_DIFFMU: The condition variable is being waited on by another task using a different muid.	cv_wait
0x84	ERR_DIFFNODE: The mutex and condition variable were created on different nodes.	cv_wait
0x88	ERR_MUID: muid incorrect: failed validity check.	cv_wait
0x89	ERR_MUDEL: Mutex has already been deleted.	cv_wait
0x8A	ERR_MUTYPE: Expected mutex object; failed validity. The mutex is not of priority protected type.	cv_wait, mu_setceil
0x8F	ERR_TSLICE: Time slice value out of range.	t_tslice

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x90	ERR_NOTVCB: Exceeds the number of task variables as defined by the kc_ntvar entry in the pSOS+ configuration table.	t_addvar
0x91	ERR_TVEXIST: Task variable already added. The variable cannot be added twice to the same task.	t_addvar
0x92	ERR_TVNOTCREAT: No such variable for the specified task.	t_delvar
0xA0	ERR_COID: The ID passed does not refer to a registered callout; validity check failed.	co_unregister
0xA1	ERR_NOCOCB: Maximum number of callouts (set by kc_ncocb entry in pSOS+ Configuration Table) have already been registered.	co_register
0xA2	ERR_BADCOTYPE: The type argument is invalid.	co_register
0xA3	ERR_COINPROG: A callout of the type similar to the type being un-registered is in progress and hence un-registration cannot be performed immediately; this error is returned only if CO_NOWAIT flag is specified.	co_unregister
0xB0	ERR_TSDDUP: There already exists a TSD object of the same name.	tsd_create
0xB1	ERR_NOTSDCB: All entries into TSD pointer array are used up.	tsd_create
0xB2	ERR_TSDIX: Validity check failed for given TSD index.	tsd_delete,tsd_getval, tsd_info, tsd_setval
0xB3	ERR_INVTSDFLG: Invalid value for the flags parameter (incorrect combination of constants).	tsd_create
0xB8	ERR_INVKEY: Invalid value of key argument.	sys_info
0xB9	ERR_MAXREP: Too many retries at getting information, because of concurrent modification of node's object table(s).	ob_roster, t_info

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x101	ERR_IODN: Illegal device (major) number.	de_close, de_cntrl, de_init, de_open, de_read, de_write, ioj_bind, ioj_getent, ioj_lock, ioj_unbind, ioj_unlock
0x102	ERR_NODR: No driver provided.	de_close, de_cntrl, de_init, de_open, de_read, de_write
0x103	ERR_IOOP: Illegal I/O function number.	de_close, de_cntrl, de_init, de_open, de_read, de_write
0x105	ERR_DNAME: Illegal device name.	dnt_add
0x106	ERR_DNLEN: Device name too long.	dnt_add
0x107	ERR_DNTFULL: DNT is full.	dnt_add
0x108	ERR_DNMAJOR: Device major number illegal.	dnt_add
0x109	ERR_DNDUP: Duplicate device name.	dnt_add
0x110	ERR_DNNFOUND: Device name not found.	dnt_find, dnt_remove
0x113	ERR_IOJUST: Specified entry already bound.	ioj_bind
0x114	ERR_IOJFULL: I/O Jump Table is full.	ioj_bindany
0x115	ERR_NBOUND: Specified entry not bound.	de_close, de_cntrl, de_init, de_open, de_read, de_write, ioj_getent, ioj_lock, ioj_unbind, ioj_unlock
0x116	ERR_IOPROT: Entry is protected.	ioj_unbind
0x117	ERR_IONLOCK: Lock count is already 0.	ioj_unlock
0xF00	FAT_ALIGN: Region 0 must be aligned on a long word boundary.	This error originates in pSOS+ initialization.

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0xF01	FAT_OVSDA: Region 0 overflow while making system data area.	This error originates in pSOS+ initialization.
0xF02	FAT_OVOBJT: Region 0 overflow while making object table.	This error originates in pSOS+ initialization.
0xF03	FAT_OVDDAT: Region 0 overflow while making device data area table.	This error originates in pSOS+ initialization.
0xF04	FAT_OVTCB: Region 0 overflow while making task structures.	This error originates in pSOS+ initialization.
0xF05	FAT_OVQCB: Region 0 overflow while making queue structures.	This error originates in pSOS+ initialization.
0xF06	FAT_OVSMCB: Region 0 overflow while making semaphore structures.	This error originates in pSOS+ initialization.
0xF07	FAT_OVTM: Region 0 overflow while making timer structures.	This error originates in pSOS+ initialization.
0xF08	FAT_OVPT: Region 0 overflow while making partition structures.	This error originates in pSOS+ initialization.
0xF09	FAT_OVRSC: Region 0 overflow while making RSC structures.	This error originates in pSOS+ initialization.
0xF0A	FAT_OVRN: Region 0 overflow while making region structures.	This error originates in pSOS+ initialization.
0xF0C	FAT_ROOT: Cannot create ROOT task.	This error originates in pSOS+ initialization.
0xF0D	FAT_IDLE: Cannot create IDLE task.	This error originates in pSOS+ initialization.
0xF0E	FAT_CHKSUM: Checksum error.	This error originates in pSOS+ initialization.
0xF0F	FAT_INVCPU: Wrong processor type.	This error originates in pSOS+ initialization.
0xF12	FAT_ILLPKT: Illegal packet type in the received packet.	This error originates in pSOS+ initialization.

TABLE A-2 pSOS+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0xF13	FAT_MIVERIF: Multiprocessor configuration mismatch at system verify.	This error originates in pSOS+ initialization.
0xF15	FAT_NODENUM: Illegal value for mc_nodenum.	This error originates in pSOS+ initialization.
0xF16	FAT_NNODES: Illegal value for mc_nnodes.	This error originates in pSOS+ initialization.
0xF17	FAT_OVMP: Region 0 overflow while making multiprocessor structures.	This error originates in pSOS+ initialization.
0xF18	FAT_KIMAXBUF: mc_kimaxbuf too small for mc_nnodes.	This error originates in pSOS+ initialization.
0xF19	FAT_ASYNCERR: Asynchronous RSC failure.	This error originates in pSOS+ initialization.
0xF1B	FAT_DEVINIT: Error during auto-initialization of a device.	This error originates in pSOS+ initialization.
0xF20	FAT_JN2SOON: Join request denied — Node already in system.	This error originates in pSOS+ initialization.
0xF21	FAT_MAXSEQ: Join request denied — Sequence number at limit.	This error originates in pSOS+ initialization.
0xF22	FAT_JRQATSLV: Join request sent to a slave node instead of the Master node.	This error originates in pSOS+ initialization.

A.2 pROBE+ Error Codes

pROBE+ error codes are returned as an `errno`. Table A-3 lists all error codes returned by pROBE+. Each listing includes the error code's hexadecimal number, and it's mnemonic and description. The error code mnemonics are also defined in `<errno.h>`.

TABLE A-3 pROBE+ Error Codes

Hex	Mnemonic and Description
0x1F01	PROBE_FAT_INSUFFMEM: Insufficient memory allocated by TD_DATASIZE or REGION 0 is too small.
0x1F02	PROBE_FAT_PROBETSK: DBG task creation error.
0x1F03	PROBE_FAT_NO_I_EXEC: pROBE+ Interface Executive not installed.
0x1F04	PROBE_FAT_DRV_INIT_CE: Failed to initialize console I/O driver.
0x1F05	PROBE_FAT_DRV_INIT_RD: Failed to initialize Rbug I/O driver.
0x1F0E	PROBE_FAT_CHKSUM: pROBE+ Checksum error.

A.3 pHILE+ Error Codes

pHILE+ error codes are returned as function return values (rather than an `errno` variable). Table A-4 lists all error codes returned by the pHILE+ component. Each listing includes the error code's hexadecimal number, it's mnemonic and description, and the pHILE+ system calls that can return it. The error code mnemonics are also defined in `<phile.h>`.

NOTE: An asterisk (*) next to an error code's description indicates that it can represent an NFS or RPC error. Sections A.3.2 and A.3.3 beginning on page A-41 provide tables of the NFS and RPC error codes that are mapped to pHILE+ error codes.

TABLE A-4 pHILE+ Error Codes

Hex	Mnemonic and Description	System Call(s)
0x2001	E_FUNC: Invalid function number. The function number passed to pHILE+ in register D0.L does not contain a code corresponding to a valid pHILE+ system call.	`access_f, chmod_f, chown_f, fchmod_f, fchown_f, ftruncate_f, link_f, lstat_f, make_dir, read_link, symlink_f, truncate_f, utime_f`
0x2002	E_FAIL: pHILE+ failure. An internal error has been detected by the pHILE+ file system manager. Report this error condition to Integrated Systems.	Should never happen.
0x2003	E_BADVOL: Inconsistent data on volume; volume corrupted. The data structures on the volume are inconsistent with each other. This is most likely the result of a crash while the pHILE+ file system manager was writing to the volume. On MS-DOS volumes, it can also indicate that an incorrect partition number has been specified.	`change_dir, close_dir, close_f, create_f, ftruncate_f, get_fn, init_vol, make_dir, move_f, open_dir, open_f, open_fn, pcinit_vol, pcmount_vol, read_dir, read_f, remove_f, stat_f, stat_vfs, sync_vol,truncate_f, unmount_vol, write_f`

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2004	E_NPHVOL: Not configured for pHILE+ format volumes. SC_PHILE_PHILE is not included.	init_vol, mount_vol, pcinit_vol, verify_vol
0x2005	E_VINITPAR: Illegal parameters to init_vol(). The parameters specified to an init_vol() call are not consistent. One of the following problems has been detected by the pHILE+ file system manager. ■ The specified starting location of the bitmap causes the FLIST to either extend beyond the end of the volume or the end of the control block region (if the volume has been partitioned into control and data block regions.) ■ The specified starting block for the data block region (SODATA) is not on a modulo 8 boundary, or it is beyond the end of the volume. ■ The bitmap begins in blocks 0-3.	init_vol
0x2006	E_MNTFULL: Attempt to mount too many volumes. Attempt to mount more volumes than specified by the pHILE+ Configuration Table parameter fc_nmount.	cdmount_vol, mount_vol, nfsmount_vol, pcmount_vol
0x2007	E_VALIEN: Wrong volume format. The volume to be mounted is not of the correct. Either it is the wrong type, i.e. mounting an MS-DOS volume with mount_vol(), or it has not been formatted by the pHILE+ file system manager.	cdmount_vol, mount_vol, pcmount_vol, verify_vol
0x2008	E_MNTED: Volume already mounted. This error condition implies one of the following: ■ An attempt was made to mount a device that is already mounted. ■ An attempt was made to initialize a mounted volume.	cdmount_vol, control_vol cmd CONTROL_VOL_PCINIT_VOL, init_vol, mount_vol, nfsmount_vol, pcinit_vol, pcmount_vol

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2009	E_MNTOPEN: Cannot unmount volume; files open. An attempt was made to unmount a volume when one or more of its files are still open. All files must be closed before a volume can be unmounted.	unmount_vol, verify_vol
0x200A	E_DMOUNT: Volume not mounted. Attempted to reference an unmounted volume.	access_f, change_dir, chmod_f, chown_f, control_vol cmd CONTROL_VOL_CHANGED_VOL, create_f, get_fn, link_f, lseek_f, lstat_f make_dir, move_f, open_dir, open_f, open_fn, read_link, read_vol, remove_f, stat_f, stat_vfs, symlink_f, sync_vol, truncate_f, unmount_vol, utime_f, verify_vol, write_vol
0x200B	E_FNAME: Filename not found. One or more of the filenames specified in a pathname cannot be located. *	access_f, change_dir, chmod_f, chown_f, create_f, get_fn, link_f, lstat_f, make_dir, move_f, open_dir, open_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, utime_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x200C	E_IFN: Illegal pathname. The pathname as specified is illegal. Possibilities are: ■ File name exceeds 255 characters. ■ Illegal character in a filename. ■ Incorrect pathname syntax.	access_f, change_dir, chmod_f, chown_f, create_f, get_fn, link_f, lstat_f, make_dir, move_f, open_dir, open_f, open_fn, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f
0x200D	E_NDD: No default directory. A relative pathname has been entered, but the calling task has never done a change_dir() call.	access_f, change_dir, chmod_f, chown_f, create_f, get_fn, link_f, lstat_f, make_dir, move_f, open_dir, open_f, open_fn, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f
0x200E	E_FORD: Directory file expected. An ordinary file was specified where a directory file was required. Either of the following are possible: ■ A file in a pathname (except the last file) is not a directory file. ■ The filename specified on a change_dir() is not a directory file. *	change_dir, create_f, get_fn, make_dir, move_f, open_f, remove_f
0x200F	E_ASIZE: Illegal Expansion Unit. An expansion unit of zero is illegal.	create_f

A

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2010	E_NODE: Null pathname. A pathname with zero characters was passed. This error is also returned when a pathname that does not end with an actual filename has been passed to create_f() or make_dir(), or to the new filename of a move_f() call. For example, a period (.) would be a legal pathname for open_f() but not for create_f().	create_f, make_dir, move_f, remove_f,
0x2011	E_FEXIST: Filename already exists.	create_f, make_dir, move_f
0x2012	E_FLIST: Too many files on volume. Attempt to create a new file when the FLIST is full. You provide the size of the FLIST when the volume is initialized.	create_f, make_dir, move_f
0x2013	E_FOPEN: Cannot remove an open file. Attempt to remove a file that is still open.	remove_f
0x2014	E_DNE: Cannot delete directory that has files.	remove_f
0x2015	E_RO: Requested operation not allowed on this file. This error implies one of the following: ■ An attempt was made to write to or lock BITMAP.SYS, FLIST.SYS, or a directory file. ■ Attempted to remove a system file. ■ Attempted to annex to BITMAP.SYS or FLIST.SYS. ■ Attempted to write to a volume mounted with sync_mode set to SM_READ_ONLY. *	annex_f, chmod_f, chown_f, create_f, fchmod_f, fchown_f, ftruncate_f, link_f, lock_f, make_dir, move_f, remove_f, truncate_f, utime_f, write_f, write_vol
0x2016	E_DIFDEV: move_f() across volumes. The old and new pathnames specified on a move_f() call are not on the same device.	move_f
0x2017	E_NOTREE: move_f() would destroy directory-tree structure. Attempted to move a directory file to a location within its own sub-tree. If allowed, the volume's file system hierarchy would no longer be a tree structure.	move_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2018	E_OFULL: Too many files open for task. A task attempted to open more files than specified by the pHILE+ Configuration Table parameter fc_ncfile.	open_dir, open_f, open_fn,
0x2019	E_NOFCB: Too many files open in system. An open_f() attempt will exceed the maximum number of open files allowed in the system as specified by the pHILE+ Configuration Table parameter fc_nfcb.	open_dir, open_f, open_fn,
0x201A	E_FIDBIG: Invalid FID, out of range. The FID provided on a pHILE+ call has a value that could not have been returned by an open_f() call. close_dir() will return this error code only if dd_fn in the XDIR has been corrupted.	annex_f, close_dir, close_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, lock_f, lseek_f, read_f, write_f
0x201B	E_FIDOFF: Invalid file ID, file not open. The file ID provided on a pHILE+ call is not an open file.	annex_f, close_dir, close_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, lock_f, lseek_f, read_f, write_f
0x201C	E_ININFULL: Index block full. The physical size of a file cannot be increased because the file's index block is full, so that no more extent descriptors can be added to the file. This error indicates that the file is badly scattered across the device. The annex_f() call should be used to produce more contiguity in the file and reduce the number of extents. On NFS volumes, this error code is returned if a file is too big.*	annex_f, create_f, ftruncate_f, make_dir, move_f, truncate_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x201D	E_VFULL: Volume full. No more free blocks of the required type (control or data) are available on the volume. This error can occur on a write_f() call whenever the file is extended; on a create_f(), make_dir(), or move_f() if a directory must be extended; or on an annex_f() call. *	annex_f, create_f, ftruncate_f, make_dir, move_f, truncate_f, write_f
0x201E	E_BADPOS: Illegal position parameter to lseek_f(). The LSEEK position parameter must be 0, 1, or 2.	lseek_f
0x201F	E_EOF: Seek past end of file. The parameters provided to lseek_f() would position the L_ptr beyond the logical end of the file. Since the L_ptr is viewed by the pHILE+ file system manager as unsigned, this error can also occur when an lseek_f() call positions the L_ptr before the beginning of a file.	lseek_f, read_dir
0x2020	E_FFULL: File size limit exceeded. A file or directory cannot be extended since it would exceed the maximum file size.	create_f, make_dir, move_f, write_f
0x2021	E_ILLDEV: Illegal device. The major device number specified in an init_vol() or mount_vol() is larger than the maximum device number specified in the pSOS+ Configuration Table.	cdmount_vol, init_vol, mount_vol, pcmount_vol, verify_vol
0x2022	E_LOCKED: Data is locked. Attempt to access a region of a file that is locked.	ftruncate_f, lock_f, read_f, truncate_f, write_f
0x2023	E_BADFN: Illegal or unused filename. The FN passed to an open_fn() call is not legal. Either it is not within the limits of the FLIST or the corresponding FD is not in use.	fchmod_f, open_fn
0x2024	E_FMODE: Bad synchronization mode to mount_vol(). The mount_vol() synchronization *mode* must be 0, 1, 2, 3, or 4.	cdmount_vol, mount_vol, pcmount_vol

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2025	E_IDN: Illegal device name. An illegal device name was passed to a function requiring either a device or pathname as input. Either the device number was illegal (i.e., major/minor numbers out of bounds) or it contained a syntax error.	access_f, cdmount_vol, change_dir, chmod_f, chown_f, control_vol, create_f, fchmod_f, get_fn, init_vol, link_f, lstat_f, make_dir, mount_vol, move_f, nfsmount_vol, open_dir, open_f, open_fn, pcinit_vol, pcmount_vol, read_link, read_vol, remove_f, stat_f, stat_vfs, symlink_f, sync_vol,truncate_f, unmount_vol,utime_f, verify_vol,write_vol
0x2026	E_BADMS: MS-DOS volume; illegal operation. This function cannot be used with MS-DOS volumes.	access_f, annex_f, chmod_f, chown_f, fchmod_f, fchown_f, link_f, lock_f, lstat_f, read_link, symlink_f, utime_f,
0x2027	E_ILLMSTYP: Illegal DOS disk type. The pcinit_vol() parameter dktype exceeds the maximum allowable value.	pcinit_vol
0x2029	E_NMSVOL: Not configured for MS-DOS FAT format volumes. SC_PHILE_MSDOS is not included.	pcmount_vol
0x202A	E_BADGEOM: Illegal or inconsistent geometry. The values in DISK_VOLUME_GEOMETRY are out of bounds or inconsistent with each other.	pcinit_vol dktype DK_DRIVER, control_vol cmd CONTROL_VOL_PCINIT_VOL

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2030	E_ECMD: Illegal command. The cmd is illegal or not supported by the volume's file system type.	control_vol
0x2031	E_EARG: Illegal argument. The arg is not appropriate for the selected command.	control_vol cmd CONTROL_VOL_CHANGED_VOL
0x2041	E_BUFSIZE: Buffers not available for block size. The pHILE+ format volume has a block size greater than the cache buffer size. Increase fc_logbsize. However, this also increases the block size of any pHILE+ format volume initialized by init_vol().	mount_vol
0x2042	E_MEDIA: Change occurred for removable media. Attempted to reference a volume that was changed. This happens if a file descriptor refers to a volume that was changed. The file descriptor should be closed. It also happens if a volume is changed in the middle of a system call that uses the volume.	All system calls.
0x2050	E_BADNFS: NFS volume; illegal operation. This function cannot be used with NFS volumes.	annex_f, get_fn, lock_f, open_fn, read_vol, sync_vol, write_vol
0x2051	E_MAXLOOP: Symbolic links nested too deeply. A pathname contains symbolic links nested more than three levels deep.	access_f, change_dir, chmod_f, chown_f, create_f, link_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, verify_vol

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2052	E_EREMOTE: "Too many levels of remote in path" on server. *	access_f, change_dir, chmod_f, chown_f, close_dir, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2053	E_PERM: The task does not have the ownership that is needed. The task does not have ownership for the requested file operation. *	remove_f
0x2054	E_EIO: A hard error occurred at a remote site. There was some hardware error, such as an I/O error, at the server. Abort the operation. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2055	E_EACCES: The task does not have the necessary access permissions. The task does not have permission for the requested file operation. *	access_f, change_dir, chmod_f, chown_f, close_dir, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, utime_f, write_f
0x2056	E_EISDIR: Illegal operation on a directory. If you attempt an operation on a directory as if it were a data file, this error is reported.*	ftruncate_f, move_f, read_dir, read_f, remove_f, truncate_f, write_f
0x2057	E_EQUOT: Quota exceeded. The server enforces a disk usage quota for each user. If this error is reported, use the disk less, or remove files, or have the quota raised. *	create_f, ftruncate_f, link_f, make_dir, move_f, symlink_f, truncate_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2058	E_ESTALE: Stale file handle, file handle invalid. When a server crashes, or there is some other exceptional event, file handles no longer are valid. Consider unmounting the file system and remounting it. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2059	E_XLINK: Can't close link.	link_f
0x205A	E_NAMETOOLONG: Directory/filename too long.	read_dir
0x205B	E_ENXIO: "No such device or address" on server.*	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x205C	E_ENODEV: "No such device" on server. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x205D	E_NNFSVOL: Not configured for NFS client volumes. SC_PHILE_NFS is not included.	nfsmount_vol
0x2060	E_BADCD: CD-ROM volume; illegal operation.	access_f, annex_f, chmod_f, chown_f, create_f, fchmod_f, fchown_f, ftruncate_f, link_f, lock_f, lstat_f, make_dir, move_f, read_link, remove_f, symlink_f, sync_vol, truncate_f, utime_f, write_f, write_vol
0x2061	E_NCDVOL: Not configured for CD-ROM ISO 9660 format volumes. SC_PHILE_CDROM is not included.	cdmount_vol
0x2062	E_CDMVOL: Multi-volume CD-ROM not supported.	cdmount_vol
0x2063	E_CDBSIZE: Volume not made with 2K block size.	cdmount_vol

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2064	E_CDFMT: CD format not ISO 9660 compatible.	cdmount_vol
0x2070	E_EAUTH: "Authentication error" on server. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2071	E_ENFS: NFS error. "Portmap error" on server.*	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2072	E_ETIMEDOUT: NFS call timed out. A server did not respond to a request. The requested NFS call timed out. Network congestion, a server that is down, or a hardware problem may be the cause.*	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2074	E_ENOAUTHBLK: No RPC authorization blocks available.*	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2075	E_ECANTSEND: Failure in sending call. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2076	E_ECANTRECV: Failure in receiving result. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2077	E_PROGUNAVAIL: Program not available. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2078	E_EPROGVERSMISMATCH: Program version mismatched. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2079	E_ECANTDECODEARGS: Decode arguments error. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x207A	E_EUNKNOWNHOST: Unknown host name. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x207B	E_EPROGNOTREGISTERED: Remote program is not registered. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x207C	E_UNKNOWNPROTO: Unknown protocol. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x207D	E_EINTR: Call interrupted. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x207E	E_ERPC: All other RPC errors. *	access_f, change_dir, chmod_f, chown_f, create_f, fchmod_f, fchown_f, fstat_f, fstat_vfs, ftruncate_f, link_f, lseek_f, lstat_f, make_dir, move_f, nfsmount_vol, open_dir, open_f, read_dir, read_f, read_link, remove_f, stat_f, stat_vfs, symlink_f, truncate_f, unmount_vol, utime_f, write_f
0x2127	VF_INDIR: Incomplete directory entry. The last entry in a directory is a long file name and the end of the directory entry is missing.	verify_vol

TABLE A-4 pHILE+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x2200	VF_INSUFF: Insufficient working area provided. Supply more memory for the data area pointed to by pb_dataptr and increase pb_datalen. Refer to page 3-92.	verify_vol
0x2201	VF_MAXDEPTH: Maximum depth exceeded on directory traversal. Increase the value of pb_maxdepth. If needed, supply more memory for pb_dataptr and increase pb_datalen. Refer to page 3-92.	verify_vol
0x2202	VF_ABORT: Verify routine aborted by user.	verify_vol
0x2F01	FAT_NORAM: Insufficient data area.	This error originates in pHILE+ initialization.
0x2F02	FAT_HSTLNG: SC_PHILE_NFS included and local host name too long.	This error originates in pHILE+ initialization.
0x2F03	FAT_NOHST: SC_PHILE_NFS included and local host name not available.	This error originates in pHILE+ initialization.
0x2F04	FAT_NOPRPC: SC_PHILE_NFS included and pRPC+ not included.	This error originates in pHILE+ initialization.
0x2F0E	FAT_PHCSUM: Checksum error in the pHILE+ file system manager.	This error originates in pHILE+ initialization.

A.3.1 pSOS+ Errors Related to pHILE+

When a task is deleted or restarted, pSOS+ may return errors related to pHILE+. These error codes are listed in Table A-5.

TABLE A-5 pSOS+ Errors Related to pHILE+

Hex	Description	System Call(s)
0x0D	ERR_RSTFS: Possible file system corruption. A task was restarted while executing pHILE+ code, resulting in a possible inconsistency in the volume data structures. Following this error, use of `verify_vol()` on the volume is recommended.	(pSOS+) `t_restart`
0x18	ERR_DELFS: Attempt to delete task using the pHILE+ file system manager. A task that has open files or is holding pHILE+ resources cannot be deleted.	(pSOS+) `t_delete`

A

A.3.2 Conversions of NFS Error Codes

All NFS errors received by pHILE+ are mapped to pHILE+ error codes. Table A-6 shows the conversions of these codes. If an NFS error not listed below is received, it is mapped to the code E_FAIL 0x2002, “pHILE+ failure”. This should never happen, unless a new NFS error code is defined at the server.

Codes whose second-to-last digit is not 5 can also represent errors from other file systems.

TABLE A-6 pHILE+ Error Codes That Represent NFS Errors

pHILE+ Hex	pHILE+ Description	NFS Hex	NFS Description
0x200B	E_FNAME: Filename not found.	0x02	No such file or directory.
0x200C	E_IFN: Illegal pathname.	0x3f	File name too long.
0x200E	E_FORD: Directory file expected.	0x14	Not a directory.
0x2011	E_FEXIST: File already exists.	0x11	File exists.
0x2014	E_DNE: Directory not empty.	0x42	Directory not empty.

TABLE A-6 pHILE+ Error Codes That Represent NFS Errors (Continued)

pHILE+ Hex	pHILE+ Description	NFS Hex	NFS Description
0x2015	E_RO: Illegal on system or directory file.	0x1e	Read-only file system.
0x201C	E_ININFULL: Index block full.	0x1b	File too large.
0x201D	E_VFULL: Volume is full.	0x1c	No space left on device.
0x2052	E_EREMOTE: Too many levels of remote in path.	0x47	Too many levels of remote in path.
0x2053	E_PERM: Task does not have ownership.	0x01	Not owner.
0x2054	E_EIO: Hard error happened at remote site.	0x05	I/O error.
0x2055	E_EACCESS: Task does not have access permissions.	0x0d	Permission denied.
0x2056	E_EISDIR: Illegal operation on a directory.	0x15	Is a directory.
0x2057	E_EQUOT: Quota exceeded.	0x45	Disc quota exceeded.
0x2058	E_ESTALE: Stale NFS file handle.	0x46	Stale NFS file handle.
0x205B	E_ENXIO: No such device or address.	0x06	No such device or address.
0x205C	E_ENODEV: No such device.	0x13	No such device.

A.3.3 Conversions of RPC Error Codes

All RPC errors received by pHILE+ are mapped to pHILE+ error codes. Table A-7 shows the conversions of these codes. If an RPC error code not listed below is received, it is mapped to the code E_ERPC 0x207E, “All other RPC errors”.

TABLE A-7 pHILE+ Error Codes That Represent RPC Errors

pHILE+ Hex	pHILE+ Description	RPC Code	RPC Description
0x2070	E_EAUTH: RPC Authorization is not available.	7	RPC_AUTHERROR: Authentication error.
0x2071	E_ENFS: NFS error - pmapper failure.	14	RPC_PORTMAPFAILURE: The pmapper failed in its call.
0x2072	E_ETIMEDOUT: NFS call timed out.	5	RPC_TIMEDOUT: Call timed out.
0x2075	E_ECANTSEND: Failure in sending call.	3	RPC_CANTSEND: Failure in sending call.
0x2076	E_ECANTRECV: Failure in receiving result.	4	RPC_CANTRECV: Failure in receiving result.
0x2077	E_PROBUNAVAIL: Program not available.	8	RPC_PROGUNAVAIL: Program not available.
0x2078	E_EPROGVERSMISMATCH: Program version mismatched.	9	RPC_PROGVERSMISMATCH: Program version mismatched.
0x2079	E_ECANTDECODEARGS: Decode arguments error.	11	RPC_CANTDECODEARGS: Decode arguments error.
0x207A	E_EUNKNOWNHOST: Unknown host name.	13	RPC_UNKNOWNHOST: Unknown host name.
0x207B	E_PROGNOTREGISTERED: Remote program is not registered.	15	RPC_PROGNOTREGISTERED: Remote program is not registered.
0x207C	E_UNKNOWNPROTO: Unknown protocol.	17	RPC_UNKNOWNPROTO: Unknown protocol.

TABLE A-7 pHILE+ Error Codes That Represent RPC Errors (Continued)

pHILE+ Hex	pHILE+ Description	RPC Code	RPC Description
0x207D	E_EINTR: Call interrupted.	18	RPC_INTR: Call interrupted.
0x207E	E_ERPC: All other RPC errors.	1	RPC_CANTENCODEARGS: Can't encode arguments.
		2	RPC_CANTDECODERES: Can't decode results.
		6	RPC_VERSMISMATCH: RPC versions not compatible.
		10	RPC_PROCUNAVAIL: Procedure unavailable.
		12	RPC_SYSTEMERROR: Generic "other problem".
		16	RPC_FAILED

A.4 pREPC+ Error Codes

When a pREPC+ system call generates an error, an error code is loaded into an internal variable that can be read through the macro `errno()`. One `errno` variable exists for each task. If the return value of a pREPC+ system call indicates an error, your application should examine the `errno` variable to determine the cause of the error. See the description of `errno()` on page 4-44 for more information.

Table A-8 lists the error codes of the pREPC+ library and component. Each listing includes the error code's hexadecimal number, its mnemonic, and a brief description. The error code mnemonics are also defined in `<prepc.h>`.

For practical reasons, system calls are not listed, because nearly every pREPC+ error code can be returned by all pREPC+ system calls. In addition, errors in other pSOSystem components or device drivers can be reported by pREPC+ system calls.

TABLE A-8 pREPC+ Error Codes

Hex	Mnemonic and Description
0x3001	`EMOPEN`: Maximum number of files are open.
0x3002	`ERANGE`: Converted value out of range.
0x3003	`EBASE`: Invalid radix base specified.
0x3005	`EACCESS`: File access violation.
0x3006	`EMODE`: Unrecognized mode specified.
0x3007	`EINVAL`: Operation not allowed on this type of file.
0x3008	`EPHILE`: Attempted a disk file operation without the pHILE+ file system manager installed.
0x3009	`EINVTYPE`: Invalid buffer type.
0x300A	`EINVSIZE`: Invalid buffer size.
0x300B	`EPRRW`: Previous read/write; cannot setvbuf.
0x300D	`ENAN`: Invalid floating point number.
0x300E	Domain error.
0x3F01	`LC_FAT_CONFIG`: Insufficient memory to hold pREPC+ data.
0x3F03	`LC_FAT_STDIO`: Cannot open standard I/O streams.
0x3F0E	`LC_FAT_CHKSUM`: Corrupted ROM; checksum error.
0x5060	Converted or resultant value is out of range.

A.5 pLM+ Error Codes

When the pLM+ library manager generates an error, an error code is loaded into an internal variable that can be read through the macro `errno()`. One `errno` variable exists for each task. If the return value of a pLM+ system call indicates an error, your application should examine the `errno` variable to determine the cause of the error. See the description of `errno()` on page 4-44 for more information.

Table A-9 lists the error codes of the pLM+ library manager. Each listing includes the error code's hexadecimal number, its mnemonic and description, and the system calls that can return it. The error code mnemonics are also defined in the file `<errno.h>`.

TABLE A-9 pLM+ Error Codes

Hex	Mnemonic and Description	System Call(s)
0x4001	`LME_FUNC`: Invalid function number.	
0x4002	`LME_NOTFOUND`: Library with the given name is not found.	`sl_bindindex`, `sl_getindex`
0x4003	`LME_MAJORVER`: The currently registered library has a different major version if `scope` is `SL_EXACT` or `SL_COMP` for the specified library.	`sl_acquire`, `sl_bindindex`, `sl_getindex`
0x4004	`LME_MINORVER`: The currently registered library has a different minor version if scope is `SL_EXACT` for the specified library. If scope is `SL_COMP`, the currently registered revision has a lower minor version. In both cases, the currently registered library is of the expected major version.	`sl_acquire`, `sl_bindindex`, `sl_getindex`
0x4005	`LME_INVSCOPE`: Invalid value in the variable `scope`.	`sl_acquire`, `sl_getindex`
0x4011	`LME_NOTLIB`: Not a shared library at specified address returned by the `LM_LOADCO` callout for the specified library.	`sl_acquire`, `sl_register`, `sl_update`
0x4012	`LME_MAXREG`: The registered library table for pLM+ is full.	`sl_acquire`, `sl_register`
0x4013	`LME_DUPLIB`: Another registered library has the same name.	`sl_register`

TABLE A-9 pLM+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x4014	LME_KEYCOLL: A registered library with a different name has the same key.	sl_acquire, sl_register
0x4015	LME_REGEV: The entry function of the specified library returned a non-zero value during the SL_REGEV event.	sl_acquire, sl_register
0x4016	LME_INVMODE: Invalid unload mode attribute.	sl_register, sl_setattr, sl_update
0x4017	LME_NOLOADCO: No load call-out.	
0x4018	LME_REENTER: Re-entered pLM+ inside a call-out or entry function.	
0x4021	LME_BIGINDEX: The index is not within the legal range.	sl_getattr, sl_getsymaddr, sl_release, sl_setattr, sl_unregister, sl_update
0x4022	LME_INVINDEX: The index does not refer to a registered library.	sl_getattr, sl_getsymaddr, sl_release, sl_setattr, sl_unregister, sl_update
0x4023	LME_LIBINUSE: The library is in use by some process.	sl_unregister
0x4024	LME_UNREGEV: The entry function of the specified library returned a non-zero value during the SL_UNREGEV event. This denotes information only.	sl_release, sl_unregister
0x4025	LME_NOUNLDCO: No unload call-out.	
0x4031	LME_NOSYMTBL: The shared library does not contain an exported symbol table.	sl_getsymaddr
0x4032	LME_NOSYM: The symname symbol cannot be found in the specified library.	sl_getsymaddr

TABLE A-9 pLM+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x4041	LME_ATTACHEV: The entry function of the specified library returned a non-zero value during the SL_ATTACHEV event.	sl_acquire
0x4042	LME_LOADCO: A non-zero value returned by the LM_LOADCO callout for the specified library.	sl_acquire
0x4043	LME_DIFFLIB: A library with different name and/or version other than the one requested is returned by the LM_LOADCO callout for the specified library. (The LM_LOADCO program has a bug.)	sl_acquire, sl_update
0x4051	LME_NOTACQ: The calling process has not acquired the shared library of specified index.	sl_release
0x4052	LME_DETACHEV: The entry function of the specified library returned a non-zero value during the SL_DETACHEV event. This denotes information only.	sl_release
0x4053	LME_UNLDCO: A non-zero value returned by the LM_UNLOADCO callout for the specified library. This denotes information only.	sl_release
0x4061	LME_DIFFDEPLIB: Both new and current libraries do not have the same first level dependent library names in their headers. This error occurs only if the current library has been acquired by some process.	sl_update

A.6 pNA+ Error Codes

When the pNA+ network manager generates an error, an error code is loaded into an internal variable that can be read through the macro `errno()`. One `errno` variable exists for each task. If the return value of a pNA+ system call indicates an error, your application should examine the `errno` variable to determine the cause of the error. See the description of `errno()` on page 4-44 for more information.

Table A-10 lists the error codes of the pNA+ network manager. Each listing includes the error code's hexadecimal number, its mnemonic and description, and the system calls that can return it. The error code mnemonics are also defined in the file `<pna.h>`.

TABLE A-10 pNA+ Error Codes

Hex	Mnemonic and Description	System Call(s)
0x5006	ENXIO: No such address.	ioctl
0x5009	EBADS: The socket descriptor is invalid.	accept, bind, close, connect, getpeername, getsockname, getsockopt, ioctl, recv, recvfrom, recvmsg, select, send, sendmsg, sendto, setsockopt, shr_socket, shutdown
0x500D	E_ACCESS: Permission denied. Permission is denied because the broadcast option is not set for this socket.	send, sendmsg, sendto
0x5011	EEXIST: Duplicate entry exists.	ioctl

TABLE A-10 pNA+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x5016	EINVALID: An argument is invalid.	accept, add_ni, bind, close, connect, ioctl, listen, recv, recvfrom, recvmsg, send, sendmsg, sendto, setsockopt, shutdown, socket
0x5017	ENFILE: An internal table has run out of space.	accept, add_ni, shr_socket, socket
0x5020	EPIPE: The connection is broken.	send, sendmsg, sendto
0x5023	EWOULDBLOCK: This operation would block, but socket is non-blocking.	accept, recv, recvfrom, recvmsg, send, sendmsg, sendto
0x5024	EINPROGRESS: The socket is non-blocking, and the connection cannot be completed immediately.	connect
0x5025	EALREADY: The socket is non-blocking, and a previous connection attempt has not yet been completed.	connect
0x5027	EDESTADDRREQ: The destination address is invalid.	sendmsg, sendto
0x5028	EMSGSIZE: Message too long. The data cannot be transmitted as a unit.	recvmsg, send, sendmsg, sendto
0x5029	EPROTOTYPE: Wrong protocol type for socket.	socket
0x502A	ENOPROTOOPT: Protocol option not available (that is, the optname or level is not valid).	getsockopt, setsockopt
0x502B	EPROTONOSUPPORT: Protocol not supported (that is, the protocol argument is not valid).	socket

TABLE A-10 pNA+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x502D	EOPNOTSUPP: Requested operation not valid for this type of socket.	accept, ioctl, listen
0x502F	EAFNOSUPPORT: Address family not supported (that is, the sin_family member of sockaddr_in is not AF_INET).	connect
0x5030	EADDRINUSE: Address is already in use.	bind, listen, setsockopt
0x5031	EADDRNOTAVAIL: Address not available. For setsockopt, the multicast address was not available because of one of the following: the multicast address was not found, the interface could not be determined, or the interface does not support multicast.	bind, connect, listen, sendmsg, sendto, setsockopt
0x5033	ENETUNREACH: Network is unreachable.	send, sendmsg, sendto
0x5035	ECONNABORTED: The connection has been aborted by the peer.	accept
0x5036	ECONNRESET: The connection has been reset by the peer.	recv, recvfrom, recvmsg, send, sendmsg, sendto
0x5037	ENOBUFS: An internal buffer is required but cannot be allocated (that is, insufficient resources are available. For ioctl, insufficient resources are available to install a new route.).	accept, connect, getpeername, getsockname, ioctl, recv, recvfrom, recvmsg, send, sendmsg, sendto, setsockopt, socket
0x5038	EISCONN: The socket is already connected.	connect, sendmsg, sendto

TABLE A-10 pNA+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x5039	ENOTCONN: The socket is not connected.	getpeername, recv, recvfrom, recvmsg, send, sendmsg, sendto, shutdown
0x503B	ETOOMANYREFS: Too many references: can't splice. The per socket maximum number of memberships has been exceeded.	setsockopt
0x503C	ETIMEDOUT: Connection timed out.	connect
0x503D	ECONNREFUSED: The attempt to connect was refused.	connect, listen
0x5041	EHOSTUNREACH: The destination host could not be reached from this node.	send, sendmsg, sendto
0x5046	ENIDOWN: NI_INIT returned -1.	add_ni
0x5047	ENMTU: The MTU is invalid.	add_ni
0x5048	ENHWL: The *hardware* length is invalid.	add_ni
0x5049	ENNOFIND: The route specified cannot be found.	
0x504A	ECOLL: Collision in select call; these conditions have already been selected by another task.	select
0x504B	ETID: The task ID is invalid.	ioctl, shr_socket
0x5F01	FAT_INSUFFMEM: Insufficient memory allocated by nc_datasize or Region 0 too small; increase nc_datasize of Region 0 or reduce the number of required data structures specified in the pNA+ Configuration Table.	This error originates in pNA+ initialization.
0x5F02	FAT_NRT: The number of initial routing table entries specified exceeds nc_nroute. Increase nc_nroute.	This error originates in pNA+ initialization.
0x5F03	FAT_NNI: The number of initial NI table entries specified exceeds nc_nni. Increase nc_nni.	This error originates in pNA+ initialization.

TABLE A-10 pNA+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x5F04	FAT_NIHSIZE: Invalid NI address.	This error originates in pNA+ initialization.
0x5F05	FAT_NIMTU: Invalid MTU for NI.	This error originates in pNA+ initialization.
0x5F06	FAT_PNAMEM: pNA+ memory error.	This error originates in pNA+ initialization.
0x5F07	FAT_PNATASK: PNAD task creation error.	This error originates in pNA+ initialization.
0x5F08	FAT_PNAINIT: pNA+ initialization error.	This error originates in pNA+ initialization.
0x5F09	FAT_NIINIT: NI initialization error.	This error originates in pNA+ initialization.
0x5F0A	FAT_RTINIT: Routing table initialization error.	This error originates in pNA+ initialization.
0x5F0B	FAT_ARPINIT: ARP table initialization error.	This error originates in pNA+ initialization.
0x5F0C	FAT_TIMERINIT: PNAD timer initialization error.	This error originates in pNA+ initialization.
0x5F0D	FAT_EVENT: PNAD event error.	This error originates in pNA+ initialization.
0x5F0E	FAT_CHKSUM: pNA+ checksum error.	This error originates in pNA+ initialization.
0x5F0F	EINTERNAL: pNA+ detects an inconsistency in the control information for network resources it manages. In most cases, this error is caused by a corruption of the pNA+ data area by an errant user task. Check pNA+ data configurations and memory usage by tasks. For further advice, contact pSOSystem technical support with information on pNA+ configuration and a dump of register contents.	This error can be detected at several points within pNA+.

TABLE A-10 pNA+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x5F10	FAT_NHT: The number of initial host table entries specified exceeds nc_nhentry. Increase nc_nhentry.	This error originates in pNA+ initialization.
0x5F11	FAT_FUNC: An invalid system call code was passed to pNA+. Check the pNA bindings file in pSOSystem.	This error originates when a system call is made.

A.7 pRPC+ Error Codes

When a pRPC+ system call generates an error, an error code is loaded into an internal variable that can be read through the macro `errno()`. One `errno` variable exists for each task. If the return value of a pRPC+ system call indicates an error, your application should examine the `errno` variable to determine the cause of the error. See the description of `errno()` on page 4-44 for more information.

Table A-11 lists the error codes of the pRPC+ subcomponent. Each listing includes the error code's hexadecimal number, its mnemonic and description, and the system calls that can return it. The error code mnemonics are also defined in `<prpc.h>`.

TABLE A-11 pRPC+ Error Codes

Hex	Mnemonic and Description	System Call(s)
0x5101	FAT_PRPC_CHKSUM: Corrupted ROM.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5102	FAT_PRPC_MEM: Insufficient memory to hold pRPC+ data.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5104	FAT_PRPC_TASKCREATE: Cannot start `portmapper`.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5F41	FAT_CKSUM: checksum error.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5F42	FAT_MEMSIZE: memory not sufficient, if data size in `nr_datasize` is less than pRPC data size, or not sufficient free memory.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5F43	Not used.	
0x5F44	FAT_TASKCREATE: error while creating `portmapper` task.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.

TABLE A-11 pRPC+ Error Codes (Continued)

Hex	Mnemonic and Description	System Call(s)
0x5F45	FAT_CALLOUT: the callout functions `nr_gethostname` and `nr_get_hostbyname` are not registered.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5F46	FAT_TASKSETPRI: `portmapper` task `t_setpri()` error.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x5F47	FAT_TASKSTART: `portmapper` task start error.	This error originates in pRPC+ initialization when used as a subcomponent of pNA+.
0x6F05	E_CHKSUM: checksum error.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.
0x6FC2	FAT_MEMSIZE: memory not sufficient, if data size in `nr_datasize` is less than pRPC data size, or not sufficient free memory.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.
0x6FC4	FAT_TASKCREATE: error while creating `portmapper` task.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.
0x6FC5	FAT_CALLOUT: the callout functions `nr_gethostname` and `nr_get_hostbyname` are not registered.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.
0x6FC6	FAT_TASKSETPRI: portmapper task `t_setpri()` error.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.
0x6FC7	FAT_TASKSTART: portmapper task start error.	This error originates in pRPC+ initialization when used as a subcomponent of pSE+.

A.8 pMONT+ Error Codes

An error generated during communication can cause the ESp/pMONT+ connection to break. Examples of such an error are a queue filling up or an insufficient number of memory buffers for socket communication. pMONT+ stores the error code so that, upon the next established connection, the ESp log window displays the error code as `prevErr`. This error code can be a pMONT+, pSOS+, pNA+, or serial driver error code returned during the previous connection. The code may indicate the problem in the configuration that caused the failure.

Table A-12 summarizes the pMONT+ error codes.

TABLE A-12 pMONT Error Codes

Hex Code	Mnemonic	Description
0x7801	PMONT_ERR_TIMEOUT	Communication timed out
0x7802	PMONT_ERR_UNIMPL	Unimplemented service
0x7803	PMONT_ERR_EV_FULL	Trace buffer full
0x7F01	PMONT_FAT_NORAM	Insufficient memory
0x7F0E	PMONT_FAT_CHKSUM	Incorrect checksum

Additionally, pMONT+ may signal a fatal error (by issuing a `k_fatal()call`) if certain pSOS+ or pNA+ calls fail. D1 contains the error code, which can be in one of the following ranges:

- 0x70*nn* to 0x7*1nn* corresponds to pSOS+ error codes *nn* to `1`*nn*
- 0x75*nn* corresponds to pNA+ error code 0x50*nn*

A.9 Driver Error Codes

The tables below list the error codes returned by pSOSystem drivers. Each driver's table lists all the error codes returned by that driver. Error code information includes hexadecimal numbers, mnemonics and descriptions.

Drivers return error codes through the pSOS+ Kernel-to-Driver Interface using the `out_retval` element of the `ioparms` structure. The contents of `out_retval` are copied to the variable pointed to by the service call input parameter `retval`. The parameter `retval` is part of the Application-to-pSOS+ Interface. For example, any driver errors resulting from a `de_read()` service call to a driver are returned at the address pointed to by the `retval` argument using the following syntax:

```
err_code = de_read(dev, iopb, &retval);
```

See the chapter "I/O System" in *pSOSystem System Concepts* for more information on the pSOS+ Kernel-to-Driver Interface and the Application-to-pSOS+ Interface.

Aside from the service calls `de_init()`, `de_open()`, `de_close()`, `de_read()`, `de_write()`, and `de_cntrl()`, a system call can return a driver error code directly through the return value of the system call. For example, the pHILE+ `write_f()` system call can return the SCSI driver error code `SCSI_W_PROTECTED` (0x1050001A) if a write is attempted on a write-protected drive.

A.9.1 Shared Memory Network Interface Driver Error Codes

The error codes listed in Table A-13 are returned by the Shared Memory Network Interface (NI_SMEM) driver.

TABLE A-13 Shared Memory Network Interface Driver Error Codes

Hex	Mnemonic and Description
0x10000001	`NISMEM_FAT_IPA`: Invalid IP address.

A.9.2 Shared Memory Kernel Interface Driver Error Codes

The error codes listed in Table A-14 are returned by the Shared Memory Kernel Interface (KI_SMEM) driver.

TABLE A-14 Shared Memory Kernel Interface Driver Error Codes

Hex	Mnemonic and Description
0x10000101	KISMEM_FAT_NOPB: No packet buffers.
0x10000102	KISMEM_FAT_NOQ: No queue.
0x10000103	KISMEM_FAT_NOTSUPP: Service not supported.
0x10000104	KISMEM_FAT_BINSTALL: Can't install bus error handler.
0x10000105	KISMEM_FAT_NOD: Number of nodes greater than maximum.

A.9.3 Terminal Interface Driver Error Codes

The error codes listed in Table A-15 are returned by the terminal interface driver.

TABLE A-15 Terminal Interface Driver Error Codes

Hex	Mnemonic and Description
0x10010200	TERM_HDWR: Hardware error.
0x10010201	TERM_MINOR: Invalid minor device.
0x10010203	TERM_BAUD: Invalid baud rate.
0x10010204	TERM_NINIT: Driver not initialized.
0x10010205	TERM_DATA: Cannot allocate data area.
0x10010206	TERM_SEM: Semaphore error.
0x10010210	TERM_AINIT: Console already initialized.
0x10010211	TERM_CHARSIZE: Bad character size.
0x10010212	TERM_BADFLAG: Flag not defined.
0x10010213	TERM_NHWFC: Hardware flow not supported.
0x10010214	TERM_BRKINT: Terminated by a break character.
0x10010215	TERM_DCDINT: Terminated by a loss of DCD.

TABLE A-15 Terminal Interface Driver Error Codes (Continued)

Hex	Mnemonic and Description
0x10010216	TERM_NBUFF: No buffers to copy characters (allocb failed).
0x10010217	TERM_NOPEN: Minor device not opened.
0x10010218	TERM_AOPEN: Channel already opened.
0x10010219	TERM_ADOPEN: Channel already opened by another driver.
0x10010220	TERM_CFGHSUP: Hardware does not support channel as configured.
0x10010221	TERM_OUTSYNC: Out of sync with DISI.
0x10010222	TERM_BADMIN: MinChar is greater than RBuffSize.
0x10010223	TERM_LDERR: Lower driver error may be corrupted structure.
0x10010224	TERM_QUE: Queue error.
0x10010225	TERM_RXERR: Data receive error.
0x10010226	TERM_TIMEOUT: Timer expired for read or write.
0x10010227	TERM_CANON: CANON and MinChar and/or MaxTime set (can only have CANON or have MinChar and/or MaxTime).
0x10010228	TERM_ROPER: Redirect operation error.
0x10010229	TERM_MARK: Received a SIOCMARK.
0x10010230	TERM_FRAMING: Framing error.
0x10010231	TERM_PARITY: Parity error.
0x10010232	TERM_OVERRUN: Overrun error.
0x10010233	TERM_NMBLK: No buffer headers (esballoc failed).
0x10010234	TERM_TXQFULL: Transmit queue is full (is returned only if WNWAIT is set).
0x10010235	TERM_NWNCONF: MaxWTime and WNWAIT both set.
0x10010236	TERM_BADCONSL: Bad default console number.
0x10010237	TERM_WABORT: Write was aborted.
0x10010238	TERM_NOMEM: Not enough memory on region zero.

A.9.4 Tick Timer Driver Error Codes

The error codes listed in Table A-16 are returned by the tick timer driver.

TABLE A-16 Tick Timer Driver Error Codes

Hex	Mnemonic and Description
0x10020001	TIMR_TICKRATE: Unsupported rate for kc_ticks2sec.

A.9.5 RAM Disk Driver Error Codes

The error codes listed in Table A-17 are returned by the RAM disk driver.

TABLE A-17 RAM Disk Driver Error Codes

Hex	Mnemonic and Description
0x10040001	RDSK_BLOCK: Block number too large.
0x10040002	RDSK_SEM: Semaphore error.
0x10040003	RDSK_MEM: Memory error.
0x10040004	RDSK_UKC: Unknown de_cntrl() function.

A.9.6 TFTP Driver Error Codes

The error codes listed in Table A-18 are returned by the TFTP driver.

TABLE A-18 TFTP Driver Error Codes

Hex	Mnemonic and Description
0x10060001	TFTP_PROTO: Protocol error detected, such as receipt of a non-DATA packet or lack of an expected message acknowledgment.
0x10060002	TFTP_TMOUT: TFTP client timed out while waiting for a response from the TFTP server.
0x10060003	TFTP_SYNC: TFTP server out of sync with TFTP client.
0x10060004	TFTP_NOSPC: No more free socket IDs.
0x10060005	TFTP_INVAL: Channel number (minor number) exceeds the maximum number of channels the driver can open.

TABLE A-18 TFTP Driver Error Codes (Continued)

Hex	Mnemonic and Description
0x10060006	TFTP_NOINIT: Call failed because the TFTP driver has not been initialized and must be initialized before the call can be made.
0x10060007	TFTP_NO_IOPB: Call failed because no IOPB was provided to de_open.
0x10060008	TFTP_NO_PATHNAME: Call failed because the pathname of the file to be transferred was not provided.
0x10060009	TFTP_NO_IPADDR: Call failed because the IP Address of the host, from which to transfer the file, was not provided.
0x1006000A	TFTP_NO_CHANNEL: Call failed because the channel being opened is either invalid, or there is an available channel for a clone open request (that is, channel=-1).

A.9.7 IDE Driver Error Codes

The error codes listed in Table A-19 are returned by the IDE driver.

TABLE A-19 IDE Driver Error Codes

Hex	Mnemonic and Description
0x10090001	IDE_HDWR: Hardware error.
0x10090002	IDE_MINOR: Invalid minor device.
0x10090003	IDE_CTRL: Invalid function code for IDE_Ctrl().
0x10090004	IDE_NINIT: Device not initialized.
0x10090005	IDE_DATA: Unable to allocate driver data area.
0x10090006	IDE_SEM: Semaphore error.
0x10090007	IDE_BBLK: Bad block.
0x10090008	IDE_UCOR: Uncorrectable error.
0x10090009	IDE_SNF: Sector not found.
0x1009000a	IDE_TONF: Track 0 not found.
0x1009000b	IDE_NDAM: Data address mark not found.

TABLE A-19 IDE Driver Error Codes (Continued)

Hex	Mnemonic and Description
0x1009000c	IDE_RANGE: Block range error.
0x1009000d	IDE_WPROT: Partition write protected
0x1009000e	IDE_NO_DEVICE: No device found
0x1009000f	IDE_PART_NUM_BAD: Bad partition number
0x10090010	IDE_INIT_DONE: Drive was already initialized
0x10090011	IDE_UKC: Unknown de_ctnrl() function
0x1009F000	IDE_DRV: Drive-related error in the last byte.

A.9.8 FLP Driver Error Codes

The error codes listed in Table A-20 are returned by the FLP driver.

TABLE A-20 FLP Driver Error Codes

Hex	Mnemonic and Description
0x100A0001	FLP_MINOR: Invalid minor device.
0x100A0002	FLP_NINIT: Device not initialized.
0x100A0003	FLP_SEM: Semaphore error.
0x100A0004	FLP_QUEUE: Cannot create a message queue.
0x100A0005	FLP_TASK: Cannot create the motor control task.
0x100A0006	FLP_RD: Floppy drive read failure.
0x100A0007	FLP_WR: Floppy drive write failure.
0x100A0008	FLP_DATA: Unable to allocate driver data area.
0x100A0009	FLP_DSKCHG: Floppy disk drive changed.
0x100AF000	FLP_DRV: Drive-related error in the last byte.

A.9.9 HTTP Driver Error Codes

The error codes listed in Table A-21 are returned by the HTTP driver.

TABLE A-21 HTTP Driver Error Codes

Hex	Mnemonic and Description
0x100C0001	HTTP_NOT_OPEN: Attempt to do I/O without first opening the I/O channel.
0x100C0002	HTTP_NO_IOPB: No IOPB provided for open or read operation.
0x100C0003	HTTP_NO_CHANNEL: The channel being opened is either invalid, or there is an available channel for a clone open request (i.e. channel=-1).
0x100C0004	HTTP_NO_INIT: Attempt to perform I/O without initializing the device.
0x100C0005	HTTP_NO_IPADDR: Either no IP address was provided or an invalid IP address was passed to open.
0x100C0006	HTTP_NO_PATHNAME: Pathname of the resource to be accessed was not provided.

A.9.10 Pipe Driver Error Codes

The error codes listed in Table A-22 are returned by the pipe driver.

TABLE A-22 Pipe Driver Error Codes

Hex	Mnemonic and Description
0x100D0001	PIPE_NOCHAN: No pipe channel available.
0x100D0002	PIPE_EMPTY: The pipe is empty.
0x100D0003	PIPE_FULL: The pipe is full.
0x100D0004	PIPE_CHAN_INVAL: The pipe channel is invalid.
0x100D0005	PIPE_MODE_INVAL: The pipe mode is invalid.
0x100D0006	PIPE_FCODE_INVAL: The pipe function code is invalid.
0x100D0007	PIPE_NO_IOPB: No IOPB was specified.

A.9.11 RDIO Driver Error Codes

The error codes listed in Table A-23 are returned by the RDIO driver.

TABLE A-23 RDIO Driver Error Codes

Hex	Mnemonic and Description
0x100E0001	RDIO_MU_FAIL: Mutex allocation failed.
0x100E0002	RDIO_MU_INV: Mutex is not created.

A.9.12 LAN Driver Error Codes

The error codes listed in Table A-24 are returned by the LAN driver.

TABLE A-24 LAN Driver Error Codes

Hex	Mnemonic and Description
0x10400001	LAN_DEVICE_NOT_SUPPORTED: The specified LAN device is not supported.
0x10400002	LAN_DEVICE_NOT_FOUND: The specified LAN device was not found.
0x10400003	LAN_CHANNEL_OVERFLOW: The LAN channel overflowed.
0x10400004	LAN_SETUP_NOT_DONE: There were problems with the LAN setup.
0x10400005	LAN_SETUP_FAILED: There were problems with the LAN setup.

A.9.13 SCSI Driver Error Codes

The error codes listed in Table A-25 are returned by the SCSI driver.

TABLE A-25 SCSI Error Codes

Hex	Mnemonic and Description
0x10500003	SCSI_FSC: Failed to send SCSI command.
0x10500009	SCSI_CHP: Failed to init SCSI chip.
0x1050000B	SCSI_UKC: Unknown de_ctnrl() function.
0x1050000C	SCSI_ID_ERR: Bad SCSI ID de_ctnrl().
0x1050000D	SCSI_NULL_CDB: NULL SCSI Control block.
0x1050000E	SCSI_PTR_CONFLICT: Both in and out data length given.
0x1050000F	SCSI_DATA_PTR_NULL: NULL data pointer.
0x10500010	SCSI_NOT_INIT: SCSI Driver not initialized.
0x10500011	SCSI_ILLRECON: Bad reconnection (no disconnect).
0x10500012	ESDNOTTDIR: Incorrect device type for operation.
0x10500013	ESDNODEVICE: No such device on SCSI bus.
0x10500014	ESBLOCKOUTOFRANGE: Block given is beyond end of disk.
0x10500015	ESODDBLOCK: Block size is less than physical.
0x10500016	ESNO_CAPACITY: Capacity shows 0 (floppy not in drive).
0x10500017	SCSI_FORMAT_FAILED: Format command failed.
0x10500018	SCSI_PART_NUM_BAD: Bad partition number.
0x10500019	SCSI_NOT_PARTITION: Device not partitioned.
0x1050001A	SCSI_W_PROTECTED: Device is write-protected.
0x1050001B	SCSI_NO_MEM: Need memory to complete command not available.
0x1050001C	SCSI_NOT_OPEN: SCSI device not open.
0x1050001D	ESDALLREADYOPEN: SCSI device is already open.
0x1050001E	SCSI_END_OF_FILE: End of tape file encountered.
0x1050001F	SCSI_MINOR_NUM_BAD: Bad minor number.
0x10500020	SCSI_ADAPTOR_NUM_BAD: Bad adapter number.

TABLE A-25 SCSI Error Codes (Continued)

Hex	Mnemonic and Description
0x10500021	SCSI_MUXFULL: SCSI MUX is full.
0x10500022	SCSI_DRVINUSE: SCSI driver is in use.
0x10500023	SCSI_DRVBADTYPE: Bad SCSI driver to load.
0x10510000	SCSI_ERR: General SCSI error code SCSI_ERR will be ORed with actual SCSI error code.
0x10510001	STAT_CHECKCOND: Target wants to give some info.
0x10510002	STAT_ERR: SCSI error that may be retried.
0x10510003	STAT_TIMEOUT: Target selection timed out.
0x10510004	STAT_BUSY: Target busy try again.
0x10510005	STAT_SEMFAIL: Semaphore call failed.
0x10510006	STAT_NOMEM: No memory available for request.
0x10510007	STAT_RETRYEXC: Failed after allotted retries.
0x10510008	STAT_RESET: SCSI bus reset (should retry).
0x10510009	STAT_BADSIZE: Drive shows no blocks.
0x1051000A	STAT_NOMEDIA: Removable disk not in drive.
0x1051000B	STAT_BLANK: End of recorded data.
0x1051000C	STAT_BAD_CMD: Target reports “Illegal Request.”
0x1051000D	STAT_NO_SENSE: Request sense returned no sense.
0x1051000E	STAT_GOT_SENSE: Sense information returned (for chip doing automatically sense).
0x1051000F	STAT_MEDCHNG: Media changed.
0x10510010	STAT_NO_CTRL: No SCSI-adapter/controller found.
0x10510011	STAT_NO_MEM: Allocate memory fails.
0x10510012	STAT_NOT_INIT: Low-level driver not initialized.

A.9.14 Parallel Driver Error Codes

The error codes listed in Table A-26 are returned by the parallel driver.

TABLE A-26 Parallel Driver Error Codes

Hex	Mnemonic and Description
0x10600400	PAR_NINIT: Channel not initialized.
0x10600401	PAR_MINOR: Invalid minor number.
0x10600402	PAR_ADOPEN: Channel already opened.
0x10600403	PAR_OUTSYNC: Dependent and independent parts of the driver out of sync.
0x10600404	PAR_SEM: Semaphore error.
0x10600405	PAR_NBUFF: No buffers available.
0x10600406	PAR_NOPEN: Channel not opened.
0x10600407	PAR_NCONNECTED: Channel not connected.
0x10600408	PAR_TIMEOUT: Timer expired for read/write.
0x10600409	PAR_CFGERROR: Configuration error.
0x10600410	PAR_TXERROR: Transmission error.
0x10600411	PAR_NMBLK: No buffer headers.
0x10600412	PAR_MODEERR: Port mode error.
0x10600413	PAR_CMDERR: Undefined command issued.
0x10600414	PAR_QUE: Could not create queue.
0x10600415	PAR_TXQFULL: Transmit queue is full.

Index

A

abort 2-79, 4-9
aborting a task 4-9
abs 4-10
absolute value
  computing 4-10, 4-48, 4-125
accept 6-5
accepting a connection on a socket 6-5
accessibility of a file 3-7
accessing global variables 7-7, 7-8
access_f 3-7
acos 4-11
acquiring a semaphore token 2-193
acquiring a shared library 5-3
acronyms xxiv
actlen 2-198
adding a network interface 6-7
adding device name to DNT 2-55
adding shared library to pLM+ table 5-17
adding task variable to task 2-201
addr 6-6, 6-10, 6-13, 6-15, 6-17
address
  converting external to internal 2-83
  converting internal to external 2-85
addrlen 6-6, 6-10, 6-13, 6-15, 6-17
add_ni 6-7
allocating a message block 6-42
allocating blocks to a file 3-8
allocating memory 4-31, 4-141, 4-178
  to a task 2-172
alloc_size 3-8
annex_f 3-8
announcing clock tick to pSOS+ kernel 2-256
appending characters to a string 4-218
appending two strings 4-205
arc cosine
  computing 4-11
arc sine
  computing 4-16
arc tangent
  computing 4-19, 4-20
arg 2-20, 3-20, 4-253, 4-255, 4-257, 6-23
arguments
  see individual argument names and individual system call names
array
  searching 4-29
  sorting 4-173
asctime 4-12
asctime_r 4-14
asin 4-16
asiz 2-168

ASR
returning from 2-16
specifying 2-9
assert 4-17
as_catch 2-9
as_notify 2-14
as_return 2-16
as_send 2-18
atan 4-19
atan2 4-20
atexit 4-22
atof 4-23
atoi 4-25
atol 4-27
attaching message block to data buffer 6-44
attr 5-11, 5-21, 5-29, 5-35
awakening tasks 2-25, 2-29

B

backlog 6-41
base 4-29, 4-173, 4-234, 4-236
bcount 3-41, 3-69, 3-73, 3-107, 3-109
bind 6-9
binding address to socket 6-9
binding an I/O jump table entry and a driver 2-69, 2-71
blkcount 3-8
block 3-73, 3-97, 3-109
message
allocating 6-42
attaching to data buffer 6-44
freeing 6-46
freeing all 6-47
blocking the calling task 2-258, 2-260
bp 6-46, 6-47
breaking number into fraction and power of 2 4-81
broadcasting identical messages to a message queue 2-128
bsearch 4-29
bsize 2-105
buf 2-37, 2-94, 2-114, 2-137, 2-157, 2-176, 2-189, 2-217, 2-272, 3-28, 3-31, 3-45, 3-67, 3-71, 3-76, 3-80, 4-14, 4-39, 4-187, 4-194, 6-49, 6-51, 6-60, 6-64
bufaddr 2-110, 2-116
buffer 2-198, 3-69, 3-73, 3-107, 3-109, 6-44
flushing 4-53
management
Transmit option 8-7
writing formatted output to 4-257
buflen 2-198, 4-14, 4-39
bufsize 3-71
buf_len 2-161

C

calling of pmap_start() 7-1
calloc 4-31
callout handler
registering 2-20
unregistering 2-23
calls. See individual call names.
calls. See system calls.
canceling armed timer 2-241
category 4-191
cdmount_vol 3-10

CD-ROM
mounting 3-10
ceil 4-33
change_dir 3-12
changing a stream buffer 4-187
changing a task execution mode 2-221
changing current directory 3-12
changing date and time 2-250
changing mode of a file 3-14, 3-25
changing mutex ceiling priority 2-98
changing owner or group of a file 3-16, 3-27
changing size of a file 3-33, 3-86
changing stream buffering characteristics 4-194
changing task priority 2-229
changing task timeslice quantum 2-239
changing timeout values 7-5
character
converting multibyte character to wide character equivalent 4-146
converting to lowercase 4-247
converting to uppercase 4-249
converting wide character into multibyte character 4-261
converting wide character string into multibyte character string 4-259
copying characters 4-148, 4-154, 4-156
determining number of bytes in 4-142
searching for 4-150
searching string for 4-206
testing for a graphical character 4-111
testing for alphabetic 4-103, 4-105
testing for control 4-107
testing for printable 4-115
testing for punctuation 4-117
ungetting 4-251
CHAR_MAX 4-131
checking status of multiple sockets 6-57
chmod_f 3-14
chown_f 3-16
clearerr 4-34
clearing error indicators 4-34
clnt 7-5
clnt_control 7-5
clnt_pcreateerror() 7-8
clnt_spcreateerr() 7-8
close 6-11
close_dir 3-17
close_f 3-18
closing
a directory file 3-17
a socket descriptor 6-11
a stream 4-49
an I/O device 2-43
an open file 3-18
cmd 3-20, 6-23
coid 2-20, 2-23
compar 4-173
comparing characters in two strings 4-220
comparing objects 4-152
comparing strings 4-207, 4-208
compatibility with older releases of pRPC+ 7-1
compute x raised to y 4-165
computing
absolute value 4-10, 4-48, 4-125

arc cosine 4-11
arc sine 4-16
arc tangent 4-19, 4-20
base-10 log 4-138
cosine 4-35
exponential function 4-47
hyperbolic cosine 4-36
hyperbolic sine 4-197
hyperbolic tangent 4-241
largest integral value 4-59
natural log 4-137
remainder 4-60
sine 4-196
smallest value 4-33
square root 4-200
tangent 4-240
computing difference between two times 4-41
connect 6-12
connection
full-duplex
terminating 6-76
control operations
performing on a volume 3-20
control_vol 3-20
CONTROL_VOL_CHANGED_VOL 3-20
CONTROL_VOL_PCINIT_VOL 3-20
conventions xx
font xx
format xxi
note, caution, and warning xxiii
revision bar xxiii
symbols xxi
converting a string
to a double 4-23, 4-228
to a long integer 4-27, 4-234
to an integer 4-25
to an unsigned long 4-236
converting broken-down time to calendar time 4-159
converting character
to lowercase 4-247
to uppercase 4-249
converting external to internal address 2-83
converting from calendar time to broken-down time 4-99, 4-101, 4-133, 4-135
converting internal address to external address 2-85
converting multibyte character string to wide character string 4-144
converting multibyte character to wide character equivalent 4-146
converting time to a string 4-12, 4-14, 4-37, 4-39
converting wide character into multibyte character 4-261
converting wide character string into multibyte character string 4-259
copying a string to another string 4-210
copying characters from one string to another 4-222
copying characters in memory 4-148, 4-154, 4-156
cos 4-35
cosh 4-36
cosine
computing 4-35
count 2-128, 2-130, 2-148, 2-182
CO_DELETE 2-20

CO_NOWAIT 2-23
co_register 2-20
CO_RESTART 2-20
CO_START 2-20
co_unregister 2-23
CO_WAIT 2-23
create_f 3-23
creating a condition variable 2-31
creating a data file 3-23
creating a directory file 3-47
creating a link between two files 3-40
creating a memory partition 2-105
creating a memory region 2-167
creating a message queue 2-130, 2-150
creating a mutex 2-87
creating a semaphore 2-182
creating a socket 6-77
creating a symbolic link to a file 3-83
creating a task 2-203
creating a temporary file 4-244
creating a TSD 2-262
ctime 4-37
ctime_r 4-39
cvid 2-25, 2-27, 2-29, 2-31, 2-33, 2-35, 2-37, 2-39, 2-41
cv_abroadcast 2-25
cv_asignal 2-27
cv_broadcast 2-29
cv_create 2-31
cv_delete 2-33
CV_FIFO 2-31
CV_GLOBAL 2-31
cv_ident 2-35
cv_info 2-37
CV_LOCAL 2-31
CV_PRIOR 2-31
cv_signal 2-39
cv_wait 2-41

D

data_area 2-47
date 2-247, 2-250, 2-254, 2-260
  changing 2-250
  putting time and date into a string 4-214
  resetting 2-254
  setting 2-254
db_input 8-3
db_output 8-5
deallocating memory 4-77
debugger
  getting input from 8-3
  sending output to 8-5
decrementing lock count of an I/O jump table entry 2-78
deleting a condition variable 2-33
deleting a file 3-75
deleting a memory region 2-170
deleting a message queue 2-133, 2-153
deleting a mutex 2-90
deleting a semaphore 2-185
deleting a task 2-208
deleting a task variable 2-211
deleting a TSD 2-266
deleting memory partition 2-108
denom 4-42, 4-128
determining number of bytes in a multibyte character 4-142

dev 2-43, 2-45, 2-47, 2-49, 2-51, 2-53
device 3-10, 3-20, 3-37, 3-49, 3-53, 3-59, 3-61, 3-65, 3-73, 3-84, 3-88, 3-91, 3-109
  adding name to DNT 2-55
  I/O
    closing 2-43
    initializing 2-47
    opening 2-49
    reading from 2-51
    requesting service 2-45
    writing to 2-53
  removing name from DNT 2-59
  returning major/minor number 2-57
devnum 2-55, 2-57
de_close 2-43
de_cntrl 2-45
de_init 2-47
de_open 2-49
de_read 2-51
de_write 2-53
diagnostic message
  printing 4-164
difftime 4-41
digit
  testing for 4-109
  testing for hexadecimal 4-123
dir 3-17, 3-55, 3-67
directory
  changing 3-12
dirname 3-55
div 4-42
division 4-42, 4-128
dktype 3-61
DK_DRIVER 3-62
DK_HARD 3-61
DK_NEC 3-61
DK_OPT 3-62
DK_12 3-61
DK_144 3-61
DK_288 3-61
DK_360 3-61
DK_720 3-61
dnt_add 2-55
dnt_find 2-57
dnt_remove 2-59
domain 6-77
driver
  binding to I/O jump table entry 2-69, 2-71
  unbinding from I/O jump table entry 2-76

E

enabling task to wait for event condition 2-64
endptr 4-228, 4-234, 4-236
entering fatal error handling mode 2-79
environment
  restoring 4-139
  saving 4-189
errno 4-44
  obtaining address of 2-60
errno_addr 2-60
errnum 4-212
error
  entering fatal error handling mode 2-79

indicators
clearing 4-34
last one reported 4-44
map error number to error message 4-212
message
printing 4-164
error codes A-1
driver A-58
FLP A-63, A-64, A-65, A-68
IDE A-62
RAM disk A-61
SCSI A-66
shared memory kernel interface A-59
shared memory network interface A-58
terminal interface A-59
TFTP A-61
tick timer A-61
origins A-2
pHILE+ A-20
conversions of NFS error codes A-41
conversions of RPC error codes A-43
pSOS+ errors related to A-41
pLM+ A-46
pMONT+ A-57
pNA+ A-49
pREPC+ A-45
pROBE+ A-19
pRPC+ A-55
pSOS+ A-4
err_code 2-79
events 2-14, 2-62, 2-64, 2-67, 2-139, 2-159, 2-191, 2-243, 2-245, 2-248
logging in pMONT trace buffer 8-6
registering 2-14, 2-139, 2-159, 2-191
sending to task 2-243, 2-245, 2-247
events_r 2-65
event_data 8-6
EV_ALL 2-65
EV_ANY 2-65
ev_asend 2-62
EV_NOWAIT 2-65
ev_receive 2-64
ev_send 2-67
EV_WAIT 2-65
exceptset 6-57
exit 4-45
exp 4-47, 4-81, 4-126
expand_unit 3-23
exponential function
computing 4-47
extent map faults 3-103
ext_addr 2-83, 2-85

F

fabs 4-48
fault descriptor 3-97
fault descriptor block 3-97
faults
extent map 3-103
fchmod_f 3-25
fchown_f 3-27
fclose 4-49
fcode 2-81
fdb_bn 3-97

fdb_code 3-97
fdb_fixable 3-97
fdb_fn1 3-97
fdb_fn2 3-97
fdb_path1 3-97
fdb_path2 3-97
feof 4-51
ferror 4-52
fflush 4-53
fgetc 4-55
fgetpos 4-56
fgets 4-57
fid 3-8, 3-18, 3-25, 3-27, 3-28, 3-33, 3-41, 3-43, 3-56, 3-59, 3-69, 3-107
file 3-31
  allocating blocks to 3-8
  changing mode of 3-14, 3-25
  changing owner or group of 3-16, 3-27
  changing size of 3-33, 3-86
  closing 3-18
  creating a temporary file 4-244
  creating link to another file 3-40
  creating symbolic link to 3-83
  data
    creating 3-23
  deleting 3-75
  determining accessibility of 3-7
  directory
    closing 3-17
    creating 3-47
    opening 3-55
  getting position indicator for 4-56, 4-91
  getting status of 3-28, 3-45, 3-76
  locking or unlocking 3-41
  moving 3-51
  obtaining number of 3-35
  opening 3-56, 3-59, 4-62
  reading from 3-69
  removing 4-180
  renaming 3-51, 4-181
  reopening 4-79
  repositioning read or write 3-43
  resetting position indicator 4-183
  setting access time 3-90
  setting modification time 3-90
  setting position indicator 4-87, 4-89
  writing to 3-107
file system
  reading directory entries 3-67
  remote
    mounting 3-53
filename 4-62, 4-79, 4-180
  generating temporary 4-245
flags 2-23, 2-31, 2-64, 2-79, 2-81, 2-87, 2-96, 2-105, 2-130, 2-141, 2-150, 2-161, 2-167, 2-172, 2-182, 2-193, 2-204, 2-263, 3-54, 6-49, 6-51, 6-55, 6-60, 6-62, 6-65
flags. See also individual flag names.
floor 4-59
flushing a buffer 4-53
fmod 4-60
fn 3-35, 3-59
fopen 4-62
forcing a task to start over 2-225
format 4-66, 4-83, 4-167, 4-185, 4-198, 4-203, 4-214, 4-253, 4-255, 4-257
fprintf 4-66

fputc 4-71
fputs 4-73
fread 4-75
free 4-77
freeing a message block 6-46
freeing all message blocks 6-47
freopen 4-79
frexp 4-81
from 6-52
fromlen 6-53
frtn 6-45
fscanf 4-83
fseek 4-87
fsetpos 4-89
fstat_f 3-28
fstat_vfs 3-31
FSTYPE_CDROM 3-32, 3-81
FSTYPE_NFS 3-32, 3-81
FSTYPE_PCDOS 3-32, 3-81
FSTYPE_PHILE 3-32, 3-81
ftell 4-91
ftruncate_f 3-33
func 2-20
function
  registering 4-22
fwrite 4-93
F_OK 3-7

G

generating random number 4-175
generating temporary filename 4-245
getc 4-95
getchar 4-96
getpeername 6-15
gets 4-97
getsockname 6-17
getsockopt 6-19
getting a task execution mode 2-221
getting address bound to socket 6-17
getting address of peer 6-15
getting address of symbol defined within library 5-15
getting attributes of library 5-11
getting buffer from a partition 2-110, 2-118
getting character
  from a stream 4-55, 4-95
  from stdin 4-96
getting file position indicator 4-91
getting group ID of task 6-14
getting identifier of a task 2-215
getting index of shared library 5-13
getting input from the debugger 8-3
getting length
  of a string 4-217, 4-226
  of a substring 4-211
getting mutex ceiling priority 2-98
getting options on a socket 6-19
getting position indicator for a file 4-56
getting statistics for a volume 3-80
getting status of a file 3-76
getting status of symbolically linked file 3-45
getting string
  from a stream 4-57
  from stdin 4-97
getting task notepad register 2-213
getting task priority 2-229
getting task timeslice quantum 2-239

getting the current time 4-242
getting tick count since initialization of system 2-252
getting timeslice quantum 2-239
getting TSD 2-274
getting TSD array entry 2-268
getting TSD index 2-270
getting user ID of task 6-14
get_fdset 7-7
get_fdset() 7-7
get_fn 3-35
get_id 6-14
gmtime 4-99
gmtime_r 4-101
group 3-16, 3-27
groupid 6-14, 6-67
groups 6-14

H

hyperbolic cosine
  computing 4-36
hyperbolic sine
  computing 4-197
hyperbolic tangent
  computing 4-241

I

IFF_BROADCAST 6-7
IFF_EXTLOOPBACK 6-7
IFF_INITDOWN 6-7
IFF_MULTICAST 6-7
IFF_NOARP 6-8
IFF_POINTTOPOINT 6-8
IFF_POLL 6-8
IFF_RAWMEM 6-8
IFF_UNNUMBERED 6-8
inbuf 8-3
inbuf_len 8-3
incrementing lock count of an I/O jump table entry 2-75
index 3-73, 3-109, 5-8, 5-10, 5-11, 5-13, 5-15, 5-21, 5-26, 5-29, 5-32, 5-35
info 7-5
initializing an MS-DOS formatted volume 3-61
initializing I/O device 2-47
initializing memory 4-158
initializing pHILE+ formatted volume 3-37
initiating connection on a socket 6-12
init_vol 3-37
integers
  dividing 4-42, 4-128
interface
  network
    adding 6-7
int_addr 2-83, 2-85
ioctl 6-23
iojent 2-69, 2-71, 2-73
ioj_bind 2-69
ioj_getent 2-73
ioj_lock 2-75
ioj_unbind 2-76
ioj_unlock 2-78
iopb 2-43, 2-45, 2-47, 2-49, 2-51, 2-53
IP level options 6-22, 6-71
ipaddr 3-54
IP_ADD_MEMBERSHIP 6-71
IP_ADD_MEMBERSHIP_INTF 6-72

IP_DROP_MEMBERSHIP 6-72
IP_DROP_MEMBERSHIP_INTF 6-73
IP_HDRINCL 6-22, 6-73
IP_MULTICAST_IF 6-22, 6-73
IP_MULTICAST_INTF 6-22, 6-73
IP_MULTICAST_LOOP 6-22, 6-74
IP_MULTICAST_TTL 6-22, 6-74
isalnum 4-103
isalpha 4-105
iscntrl 4-107
isdigit 4-109
isgraph 4-111
islower 4-113
isprint 4-115
ispunct 4-117
isspace 4-119
isupper 4-121
isxdigit 4-123
I/O jump table
  entry
    binding to driver 2-69, 2-71
    decrementing lock count of 2-78
    incrementing lock count of 2-75
    unbinding from driver 2-76
  obtaining information about entry 2-73

K

kc_nmsgbuf 8-6
key 2-197, 4-29
  PSOS_CALLOUT 2-198
  PSOS_DNT 2-198
  PSOS_IOJT 2-198
  PSOS_ROSTER 2-198
  PSOS_VERSION 2-197
k_fatal 2-79
K_GLOBAL 2-79
K_LOCAL 2-79
k_terminate 2-81

L

labs 4-125
laddr 2-105, 2-118
largest integral value
  computing 4-59
last reported error 4-44
LC_ALL 4-191
LC_COLLATE 4-191
LC_CTYPE 4-191
LC_MONETARY 4-191
LC_NUMERIC 4-191
LC_TIME 4-191
ldexp 4-126
ldiv 4-128
len 6-49, 6-51, 6-60, 6-65
length 2-105, 3-33, 3-86
letter
  testing for lowercase 4-113
  testing for uppercase 4-121
level 6-19, 6-68
libaddr 5-21, 5-35
libinfo 5-8, 5-10
libname 5-8, 5-10, 5-13
library
  getting address of symbol defined within 5-15
  getting attributes of 5-11

replacing with another of same name 5-34
search pLM+ table for 5-10
setting writable attributes of 5-28
shared
acquiring 5-3
adding to pLM+ table 5-17
getting index of 5-13
releasing 5-23
unregistering from pLM+ table 5-30
link
symbolic
reading value of 3-71
link_f 3-40
list of all pHILE+ system calls 3-2
list of all pLM+ system calls 5-2
list of all pNA+ system calls 6-2
list of all pREPC+ system calls 4-2
list of all pROBE+ system calls 8-1
list of all pRPC+ system calls 7-2
list of all pSOS+ system calls 2-2
list of all services supported by pRPC+ library 7-2
listen 6-41
listening for connections on a socket 6-41
locale 4-192
obtain current settings 4-130
obtaining or changing 4-191
localeconv 4-130
localtime 4-133
localtime_r 4-135
locking a file 3-41
locking a mutex 2-96
lock_f 3-41
log 4-137
base-10
computing 4-138
natural
computing 4-137
logging events in pMONT trace buffer 8-6
log_event 8-6
log10 4-138
longjmp 4-139
lptr 4-162
lseek_f 3-43
lstat_f 3-45
L_ptr 3-43

M

major 2-69, 2-71, 2-73, 2-75, 2-76, 2-78
make_dir 3-47
malloc 4-141
manual conventions. See conventions.
manual organization xix
map error number to error message 4-212
mask 2-221
maxlen 2-150
maxnum 2-150
maxsize 4-214
mblen 4-142
mbstowcs 4-144
mbtowc 4-146
memccpy 4-148
memchr 4-150
memcmp 4-152
memcpy 4-154
memmove 4-156

memory
allocating 4-31, 4-141, 4-178
creating partition 2-105
creating region 2-167
deallocating 4-77
deleting partition 2-108
deleting region 2-170
initializing 4-158
returning memory to a region 2-178
memset 4-158
message
broadcasting identical messages to a message queue 2-128
creating a message queue 2-130, 2-150
deleting a message queue 2-133, 2-153
obtaining ID of a message queue 2-135
posting to queue 2-120, 2-122, 2-124, 2-126, 2-144, 2-146, 2-148
querying message queue object 2-137
queue
obtaining ID of 2-155
posting message to 2-163, 2-165
querying object 2-157
requesting message from 2-161
requesting from queue 2-141
mktime 4-159
mode 2-9, 2-233, 3-7, 3-14, 3-23, 3-25, 3-47, 3-56, 3-59, 4-62, 4-79, 4-194
modf 4-162
mounting
a CD-ROM 3-10
an MS-DOS formatted volume 3-65
pHILE+ formatted volume 3-49
remote file system 3-53
mount_vol 3-49
move_f 3-51
moving a file 3-51
msg 6-54, 6-62
msg_accrights 6-55
msg_accrightslen 6-55
msg_buf 2-120, 2-122, 2-124, 2-126, 2-128, 2-142, 2-144, 2-146, 2-148, 2-161, 2-163, 2-165
MSG_DONTROUTE 6-61, 6-62, 6-65
MSG_INTERFACE 6-52, 6-65
msg_iov 6-55
msg_len 2-124, 2-126, 2-148, 2-161, 2-163, 2-165
msg_name 6-54
msg_namelen 6-54
MSG_OOB 6-49, 6-52, 6-55, 6-61, 6-62, 6-65
MSG_PEEK 6-49, 6-52, 6-55
MSG_RAWMEM 6-49, 6-52, 6-61, 6-65
muid 2-41, 2-90, 2-92, 2-94, 2-96, 2-98, 2-100
mutex 2-94
changing ceiling priority 2-98
creating 2-87
deleting 2-90
getting ceiling priority 2-98
locking 2-96
obtaining identifier of 2-92
unlocking 2-100
mu_create 2-87
mu_delete 2-90

MU_FIFO 2-88
MU_GLOBAL 2-87
mu_ident 2-92
mu_info 2-94
MU_LOCAL 2-87
mu_lock 2-96
MU_NORECURSIVE 2-87
MU_NOWAIT 2-96
MU_PRIOR 2-88
MU_PRIO_INHERIT 2-88
MU_PRIO_NONE 2-88
MU_PRIO_PROTECT 2-88
MU_RECURSIVE 2-87
mu_setceil 2-98
mu_unlock 2-100
MU_WAIT 2-96
m_ext2int 2-83
m_int2ext 2-85

N

name 2-31, 2-35, 2-55, 2-57, 2-59, 2-87, 2-92, 2-105, 2-112, 2-130, 2-135, 2-150, 2-155, 2-167, 2-174, 2-182, 2-187, 2-203, 2-215, 2-263, 2-270, 3-7, 3-12, 3-14, 3-16, 3-23, 3-35, 3-45, 3-47, 3-56, 3-71, 3-75, 3-76, 3-80, 3-86, 3-90
name1 3-40, 3-83
name2 3-40, 3-83
nbuf 2-106
nbytes 8-3
new 4-181
newname 3-51
newprio 2-98, 2-229
newval 2-274
new_mode 2-221
new_tslice 2-239
nfsmount_vol 3-53
ni 6-7
nmemb 4-29, 4-31, 4-75, 4-93, 4-173
node 2-35, 2-81, 2-92, 2-112, 2-135, 2-155, 2-187, 2-215
  non-master terminating 2-81
nptr 4-23, 4-25, 4-27, 4-228, 4-234, 4-236
number
  breaking into fraction and power of 2 4-81
  random 4-175
    setting seed for random number generator 4-201
numer 4-42, 4-128
num_of_file_ descriptors 3-38

O

objects
  comparing 4-152
obtain current locale settings 4-130
obtaining address of errno variable 2-60
obtaining ID of a message queue 2-135, 2-155
obtaining identifier
  of a condition variable 2-35
  of a mutex 2-92
  of a partition 2-112
  of a region 2-174
obtaining information
  about entry in I/O jump table 2-73
  about system 2-197
obtaining new socket descriptor 6-75
obtaining number of a file 3-35

obtaining or changing program's locale 4-191
obtaining roster of pSOS+ objects 2-102
obtaining statistics about a volume 3-31
obtaining status of a file 3-28
ob_roster 2-102
ob_type 2-102
offset 3-43, 4-87
oj_bindany 2-71
old 4-181
oldname 3-51
oldprio 2-98, 2-229
old_lptr 3-43
old_mode 2-221
old_tslice 2-239
opening a directory file 3-55
opening a file 3-56, 3-59, 4-62
opening an I/O device 2-49
open_dir 3-55
open_f 3-56
open_fn 3-59
options
  IP level 6-22, 6-71
  socket level 6-20, 6-68
  TCP level 6-21, 6-70
optlen 6-19, 6-68
optname 6-19, 6-68
optval 6-19, 6-68
output
  writing to buffer 4-198
owner 3-16, 3-27

P

paddr 2-105, 2-118
parameters
  see individual parameter names and individual system call names
params 3-37, 3-53, 3-91
partition
  getting buffer from 2-110, 2-118
  obtaining identifier of 2-112
  querying a partition object 2-114
  returning buffer to 2-116
pathname 3-54
pb_badblkptr 3-93
pb_datalen 3-92
pb_dataptr 3-92
pb_faultp 3-93
pb_fdbptr 3-93
pb_maxdepth 3-92
pcinit_vol 3-61
pcmount_vol 3-65
peer
  getting address of 6-15
Perfmeter 8-7
performing control operations
  on a socket 6-23
  on a volume 3-20
perror 4-164
pmap_start()
  calling of 7-1
pMONT
  use of pSOS+ objects 8-6
pna_allocb 6-42
pna_esballoc 6-44
pna_freeb 6-46
pna_freemsg 6-47
pos 4-89

position 3-43
posting a message to a queue 2-120, 2-122, 2-124, 2-126, 2-144, 2-146, 2-148, 2-163, 2-165
pow 4-165
pri 6-43, 6-44
printf 4-167
printing a diagnostic message 4-164
printing formatted output
  to a stream 4-66
  to stdout 4-167
prio 2-203
program
  verifying operation 4-17
prompt 8-3
prompt_len 8-3
protocol 6-77
pRPC+
  compatibility with older releases of 7-1
pSOS+
  Configuration Table 8-6
  objects
    obtaining roster of 2-102
  obtaining information about system 2-197
PSOS_CALLOUT 2-198
PSOS_DNT 2-198
PSOS_IOJT 2-198
PSOS_ROSTER 2-198
PSOS_VERSION 2-197
ptid 2-106, 2-108, 2-110, 2-112, 2-114, 2-116, 2-118
ptr 4-75, 4-77, 4-178
pt_create 2-105
PT_DEL 2-106
pt_delete 2-108
pt_getbuf 2-110
PT_GLOBAL 2-106
pt_ident 2-112
pt_info 2-114
PT_LOCAL 2-106
PT_NODEL 2-106
pt_retbuf 2-116
pt_sgetbuf 2-118
publications, related xxv
putc 4-169
putchar 4-171
puts 4-172
putting time and date into a string 4-214
pwc 4-146
pwcs 4-144, 4-259

Q

qid 2-120, 2-122, 2-124, 2-126, 2-128, 2-131, 2-133, 2-135, 2-137, 2-139, 2-141, 2-144, 2-146, 2-148, 2-150, 2-153, 2-155, 2-159, 2-161, 2-163, 2-165
qsort 4-173
querying a partition object 2-114
querying a region object 2-176
querying a semaphore object 2-189
querying a task object 2-217
querying a TSD 2-272
querying about mutex object 2-94
querying condition variable object 2-37
querying message queue object 2-137, 2-157
q_asend 2-120
q_aurgent 2-122

q_avsend 2-124
q_avurgent 2-126
q_broadcast 2-128
q_create 2-130
q_delete 2-133
Q_DEQUEUE 2-142
Q_FIFO 2-131, 2-150
Q_GLOBAL 2-130, 2-150
q_ident 2-135
q_info 2-137
Q_LIMIT 2-131
Q_LOCAL 2-130, 2-150
Q_NOLIMIT 2-131
q_notify 2-139
Q_NOWAIT 2-141, 2-161
Q_PEEK 2-142
Q_PRIBUF 2-131
Q_PRIOR 2-131, 2-150
q_receive 2-141
q_send 2-144
Q_SYSBUF 2-131
q_urgent 2-146
q_vbroadcast 2-148
q_vcreate 2-150
q_vdelete 2-153
q_vident 2-155
q_vinfo 2-157
q_vnotify 2-159
q_vreceive 2-161
q_vsend 2-163
q_vurgent 2-165
Q_WAIT 2-141, 2-161

R

rand 4-175
random number generator
  setting seed for 4-201
rand_r 4-176
read directly from a volume 3-73
read from a file 3-69
reading directory entries 3-67
reading input
  from a stream 4-75, 4-83
  from a string 4-203
  from an I/O device 2-51
  from stdin 4-185
reading value of a symbolic link 3-71
readset 6-57
read_dir 3-67
read_f 3-69
read_link 3-71
read_mask 7-7
read_vol 3-73
realloc 4-178
receiving data from a socket 6-48, 6-51, 6-54
recv 6-48
recvfrom 6-51
recvmsg 6-54
region
  obtaining identifier of 2-174
  querying object 2-176
  returning memory to 2-178
registering a callout handler 2-20
registering a function 4-22
registering events 2-14, 2-139, 2-159, 2-191

regnum 2-213, 2-231
reg_value 2-213, 2-231
releasing a semaphore token 2-180, 2-195
releasing a shared library 5-23
remainder
computing 4-60
remove 4-180
remove_f 3-75
removing a file 4-180
removing name from DNT 2-59
rename 4-181
renaming a file 3-51, 4-181
reopening a file 4-79
replacing library with another library of same name 5-34
repositioning read or write within a file 3-43
req 7-5
requesting message from message queue 2-141, 2-161
requesting service of special I/O device 2-45
reserved 3-54
resetting date and time 2-254
resetting file position indicator 4-183
resource pool
pna_allocb() 6-43, 6-44
pna_esballoc() 6-43, 6-44
specifying 6-43, 6-44
restoring environment 4-139
resultp 4-101, 4-135
resuming a suspended task 2-227
retries 3-54
returning buffer to a partition 2-116
returning device major/minor number 2-57
returning from an ASR 2-16
returning memory segment to a region 2-178
retval 2-43, 2-45, 2-47, 2-49, 2-51, 2-53
rewind 4-183
rnid 2-168, 2-170, 2-172, 2-174, 2-176, 2-178
rn_create 2-167
RN_DEL 2-168
rn_delete 2-170
RN_FIFO 2-168
rn_getseg 2-172
rn_ident 2-174
rn_info 2-176
RN_NODEL 2-168
RN_NOWAIT 2-172
RN_PRIOR 2-168
rn_retseg 2-178
RN_WAIT 2-172
rpc_createerr 7-8
rpc_createerr variable
getting access to 7-8
rpc_getcreateerr 7-8
rpc_getcreateerr() 7-8
R_OK 3-7

S

saddr 2-167
saving environment 4-189
scanf 4-185
scope 5-8, 5-13
scratchbuf 3-38

scratch_buf 3-61
search a string for tokens 4-176, 4-230, 4-232
search pLM+ table for library 5-10
searching an array 4-29
searching for characters in a string 4-206, 4-224, 4-225, 4-227
searching memory for a character 4-150
seed 4-201
  setting for random number generator 4-201
SEEK_CUR 4-87
SEEK_END 4-87
SEEK_SET 4-87
seg_addr 2-172, 2-178
select 6-57
select() 7-7
semaphore
  creating 2-182
  deleting 2-185
  object
    querying 2-189
  obtaining identifier of
    obtaining identifier of a semaphore 2-187
send 6-60
sending asynchronous signals to a task 2-18
sending data to
  a socket 6-60, 6-62
  datagram socket 6-64
  raw sockewt 6-64
sending events to a task 2-62, 2-67, 2-243, 2-245, 2-247
sending output to the debugger 8-5
sendmsg 6-62
sendto 6-64
setbuf 4-187
setjmp 4-189
setlocale 4-191
setsockopt 6-68
setting access time of a file 3-90
setting date and time 2-254
setting file position indicator 4-87, 4-89
setting modification time of a file 3-90
setting options on a socket 6-68
setting task notepad register 2-231
setting TSD 2-274
setting user ID and group ID of task 6-67
setting writable attributes of a library 5-28
setvbuf 4-194
set_id 6-67
shared library
  acquiring 5-3
  releasing 5-23
shr_socket 6-75
shutdown 6-76
signalling a condition variable 2-27, 2-39
signals 2-18
sin 4-196
sine
  computing 4-196
sinh 4-197
size 2-172, 2-263, 4-29, 4-31, 4-75, 4-93, 4-141, 4-173, 4-178, 4-194, 6-42, 6-44
sl_acquire 5-3
SL_ANY 5-8, 5-13
sl_bindindex 5-10

SL_COMP 5-8, 5-13
SL_EXACT 5-8, 5-13
sl_getattr 5-11
sl_getindex 5-13
sl_getsymaddr 5-15
sl_register 5-17
sl_release 5-23
sl_setattr 5-28
sl_unregister 5-30
sl_update 5-34
smallest value
  computing 4-33
smid 2-180, 2-182, 2-185, 2-187, 2-189, 2-191, 2-193, 2-195
sm_av 2-180
SM_BOUNDED 2-182
SM_CONTROL_WRITE 3-49, 3-65
sm_create 2-182
SM_DELAYED_WRITE 3-49, 3-65
SM_DELAY_DATE 3-49, 3-65
sm_delete 2-185
SM_FIFO 2-182
SM_GLOBAL 2-182
sm_ident 2-187
SM_IMMED_WRITE 3-49, 3-65
sm_info 2-189
SM_LOCAL 2-182
sm_notify 2-191
SM_NOWAIT 2-193
sm_p 2-193
SM_PRIOR 2-182
SM_READ_ONLY 3-10, 3-49, 3-65
SM_UNBOUNDED 2-182
sm_v 2-195
SM_WAIT 2-193
socket 6-77
  accepting a connection 6-5
  binding address to 6-9
  checking status of multiple sockets 6-57
  closing descriptor 6-11
  control operations on 6-23
  creating 6-77
  datagram
    sending data to 6-64
  getting address bound to 6-17
  getting options on 6-19
  initiating connection 6-12
  listening for connections on 6-41
  multiplexing I/O requests 6-57
  obtaining new socket descriptor for 6-75
  performing control operations on 6-23
  raw
    sending data to 6-64
  receiving data from 6-48, 6-51, 6-54
  sending data to 6-60, 6-62
  setting options on 6-68
socket level options 6-20, 6-68
socket operations 6-23
socket variables 6-31
SOCK_DGRAM 6-77
SOCK_RAW 6-77
SOCK_STREAM 6-77
sorting an array 4-173
SO_BROADCAST 6-20, 6-68
SO_DONTROUTE 6-20, 6-69
SO_ERROR 6-20

SO_KEEPALIVE 6-20, 6-69
SO_LINGER 6-20, 6-69
SO_OOBINLINE 6-20, 6-69
SO_RCVBUF 6-20, 6-69
SO_REUSEADDR 6-20, 6-69
SO_REUSEPORT 6-20, 6-69
SO_SNDBUF 6-20, 6-70
SO_TYPE 6-20
space
  testing for 4-119
specifying an ASR 2-9
sprintf 4-198
sqrt 4-200
square root
  computing 4-200
srand 4-201
sscanf 4-203
sstack 2-203
Stack Checking 8-7
starting a task 2-233
starting_bitmap_ block_number 3-38
startpos 3-41
start_addr 2-9, 2-234
start_data_block_ number 3-38
status 4-45
stat_f 3-76
stat_vfs 3-80
stdin
  getting character from 4-96
  getting string from 4-97
  reading formatted input from 4-185
stdout
  printing formatted output to 4-167
  writing character to 4-171
  writing formatted output to 4-255
strcat 4-205
strchr 4-206
strcmp 4-207
strcoll 4-208
strcpy 4-210
strcspn 4-211
stream 4-34, 4-49, 4-51, 4-52, 4-53, 4-55, 4-56, 4-57, 4-71, 4-73, 4-75, 4-79, 4-83, 4-87, 4-89, 4-91, 4-93, 4-95, 4-169, 4-183, 4-187, 4-194, 4-251, 4-253
  changing buffering characteristics 4-194
  changing the buffer 4-187
  closing 4-49
  getting character from 4-55, 4-95
  getting string from 4-57
  printing formatted output to 4-66
  reading formatted input from 4-83
  reading from 4-75
  testing end-of-file 4-51
  testing error 4-52
  writing character to 4-71, 4-169
  writing formatted output to 4-253
  writing string to 4-73, 4-172
  writing to 4-93
strerror 4-212
strftime 4-214
string
  appending characters to 4-218
  comparing characters in two strings 4-220
  converting multibyte character string to wide character string 4-144

converting to a double 4-23, 4-228
converting to a long integer 4-27, 4-234
converting to an integer 4-25
converting to an unsigned long 4-236
copying characters from one string to another 4-222
copying to another string 4-210
getting length of 4-217, 4-226
putting time and date into 4-214
reading formatted input from 4-203
search for tokens 4-176, 4-230, 4-232
searching for characters in 4-224, 4-225, 4-227
transforming for strcpm() function 4-238
strings
appending 4-205
comparing 4-207, 4-208
strlen 4-217
strncat 4-218
strncmp 4-220
strncpy 4-222
strpbrk 4-224
strrchr 4-225
strspn 4-226
strstr 4-227
strtod 4-228
strtok 4-230, 4-232
strtol 4-234
strtoul 4-236
strxfrm 4-238
substring
getting length of 4-211
suspending a task 2-237
svc_fdset 7-7
svc_fdset variable
getting access to 7-7
svc_getreqset() 7-7
symaddr 5-15
symlink_f 3-83
symname 5-15
synchronizing a volume 3-84
sync_mode 3-10, 3-49, 3-65
sync_vol 3-84
system
getting tick count since initialization 2-252
obtaining information about 2-197
system calls
file systems supported by pHILE+ calls 3-4
list of all ESp system calls 8-1
list of all pHILE+ system calls 3-2
list of all pLM+ system calls 5-2
list of all pNA+ system calls 6-2
list of all pREPC+ system calls 4-2
list of all pROBE+ system calls 8-1
list of all pRPC+ system calls 7-2
list of all pSOSystem system calls 1-2
list of all pSOS+ system calls 2-2
list of all services supported by pRPC+ library 7-2
system calls. See also individual call names.
system-wide buffer pool 8-7
sys_info 2-197
S_IEXEC 3-14, 3-25, 3-46
S_IFBLK 3-29, 3-46, 3-77
S_IFCHR 3-29, 3-46, 3-77

S_IFDIR 3-29, 3-46, 3-77
S_IFIFO 3-29, 3-46, 3-77
S_IFLNK 3-29, 3-46, 3-77
S_IFMT 3-29, 3-46, 3-77
S_IFREG 3-29, 3-46, 3-77
S_IFSOCK 3-29, 3-46, 3-77
S_IREAD 3-14, 3-25, 3-46
S_IRGRP 3-14, 3-23, 3-25, 3-29, 3-46, 3-47, 3-77
S_IROTH 3-14, 3-24, 3-25, 3-29, 3-46, 3-47, 3-77
S_IRUSR 3-23, 3-29, 3-47, 3-77
S_ISGID 3-14, 3-23, 3-25, 3-29, 3-46, 3-47, 3-77
S_ISUID 3-14, 3-23, 3-25, 3-29, 3-46, 3-47, 3-77
S_ISVTX 3-14, 3-25, 3-29, 3-46, 3-77
S_IWGRP 3-14, 3-23, 3-25, 3-29, 3-46, 3-47, 3-77
S_IWOTH 3-14, 3-24, 3-26, 3-29, 3-46, 3-47, 3-77
S_IWRITE 3-14, 3-25, 3-46
S_IWUSR 3-23, 3-29, 3-47, 3-77
S_IXGRP 3-14, 3-24, 3-25, 3-29, 3-46, 3-47, 3-77
S_IXOTH 3-14, 3-24, 3-26, 3-29, 3-46, 3-47, 3-77
S_IXUSR 3-23, 3-29, 3-47, 3-77
s1 4-148, 4-152, 4-154, 4-156, 4-176, 4-205, 4-207, 4-208, 4-210, 4-211, 4-218, 4-220, 4-222, 4-224, 4-226, 4-227, 4-230, 4-232, 4-238
s2 4-148, 4-152, 4-154, 4-156, 4-205, 4-207, 4-208, 4-210, 4-211, 4-218, 4-220, 4-222, 4-224, 4-226, 4-227, 4-230, 4-232, 4-238

T

tan 4-240
tangent
  computing 4-240
tanh 4-241
targs 2-225, 2-234
task 2-239
  aborting 4-9
  adding task variable 2-201
  allocating memory to 2-172
  awakening 2-25, 2-29
  blocking the calling 2-258, 2-260
  changing execution mode 2-221
  changing priority 2-229
  changing timeslice quantum 2-239
  creating 2-203
  creating TSD 2-262
  deleting 2-208
  deleting a TSD 2-266
  deleting task variable 2-211
  enabling to wait for event condition 2-64
  forcing to start over 2-225
  getting group ID 6-14
  getting identifier of 2-215
  getting notepad register 2-213
  getting priority 2-229
  getting TSD 2-274
  getting TSD array entry 2-268
  getting TSD index 2-270
  getting user ID 6-14
  object
    querying 2-217
  querying a TSD 2-272

resuming 2-227
sending asynchronous signals to 2-18
sending events to 2-62, 2-67, 2-243, 2-245, 2-247
setting notepad register 2-231
setting TSD 2-274
setting user ID and group ID 6-67
starting 2-233
suspending 2-237
terminating 4-45
tcount 3-69
TCP level options 6-21, 6-70
TCP_KEEPALIVE_CNT 6-21, 6-70
TCP_KEEPALIVE_IDLE 6-21, 6-70
TCP_KEEPALIVE_INTVL 6-21, 6-70
TCP_MSL 6-21, 6-70
TCP_NODELAY 6-21, 6-70
terminating a task 4-45
terminating full-duplex connection 6-76
terminating non-master node 2-81
terms, common xxiv
testing a character for alphabetic 4-103, 4-105
testing a character for control 4-107
testing for a digit 4-109
testing for a graphical character 4-111
testing for a space 4-119
testing for hexadecimal digit 4-123
testing for lowercase letter 4-113
testing for printable character 4-115
testing for punctuation character 4-117
testing for uppercase letter 4-121
testing stream end-of-file indicator 4-51
testing stream error indicator 4-52
tick
announcing to pSOS+ kernel 2-256
ticks 2-243, 2-245, 2-248, 2-250, 2-254, 2-258, 2-260
tickshi 2-252
tickslo 2-252
tid 2-18, 2-62, 2-67, 2-139, 2-159, 2-191, 2-201, 2-204, 2-208, 2-211, 2-213, 2-215, 2-217, 2-225, 2-227, 2-229, 2-231, 2-233, 2-237, 2-239, 2-268, 2-274, 6-75
time 2-247, 2-250, 2-254, 2-260, 4-242
changing 2-250
computing difference between two times 4-41
converting from calendar time 4-99, 4-101, 4-133, 4-135
converting to a string 4-12, 4-14, 4-37, 4-39
converting to calendar time 4-159
obtaining 4-242
putting time and date into a string 4-214
resetting 2-254
setting 2-254
timeo 3-54
timeout 2-23, 2-41, 2-65, 2-96, 2-142, 2-161, 2-172, 2-193, 6-58
timeout values
changing 7-5
timeptr 4-12, 4-14, 4-159, 4-215
timer 4-37, 4-39, 4-99, 4-101, 4-133, 4-135, 4-242
canceling armed timer 2-241
times 3-90
time0 4-41

time1 4-41
tmid 2-241, 2-243, 2-245, 2-248
tmpfile 4-244
tmpnam 4-245
tm_cancel 2-241
tm_evafter 2-243
tm_evevery 2-245
tm_evwhen 2-247
tm_get 2-250
tm_getticks 2-252
tm_set 2-254
tm_tick 2-256
tm_wkafter 2-258
tm_wkwhen 2-260
token
  semaphore
    acquiring 2-193
    releasing 2-180, 2-195
tolen 6-66
tolower 4-247
toupper 4-249
transforming a string for use by the strcpm() function 4-238
Transmit buffer 8-7
truncate_f 3-86
tsdix 2-266, 2-268, 2-270, 2-272, 2-274
tsdixp 2-263
TSD_ALLOC 2-263
tsd_anchorp 2-263
tsd_create 2-262
tsd_delete 2-266
tsd_getval 2-268
tsd_ident 2-270
tsd_info 2-272
TSD_INIT_CLR 2-263
TSD_INIT_COPY 2-263
TSD_INIT_NONE 2-263
TSD_NOALLOC 2-263
tsd_setval 2-274
tv_addr 2-201, 2-211
tv_value 2-201
type 2-20, 6-77
t_addvar 2-201
T_ASR 2-10, 2-221, 2-233
t_create 2-203
t_delete 2-208
t_delvar 2-211
T_FPU 2-204
t_getreg 2-213
T_GLOBAL 2-204
t_ident 2-215
t_info 2-217
T_ISR 2-10, 2-222, 2-234
T_LEVELMASKn 2-10, 2-222, 2-234
T_LEVELMASK0 2-10, 2-222, 2-234
T_LOCAL 2-204
t_mode 2-221
T_NOASR 2-10, 2-221, 2-233
T_NOFPU 2-204
T_NOISR 2-10, 2-222, 2-234
T_NOPREEMPT 2-10, 2-221, 2-233
T_NOTSLICE 2-10, 2-221, 2-233
T_PREEMPT 2-10, 2-221, 2-233
t_restart 2-225
t_resume 2-227
t_setpri 2-229
t_setreg 2-231
t_start 2-233

T_SUPV 2-10, 2-234
t_suspend 2-237
T_TSLICE 2-10, 2-221, 2-233
t_tslice 2-239
T_USER 2-10, 2-234

U

unbinding an I/O jump table entry and a driver 2-76
ungetc 4-251
ungetting a character 4-251
unit_size 2-167
unlocking
  a file 3-41
  a mutex 2-100
unmounting a volume 3-88
unmount_vol 3-88
unregistering a callout handler 2-23
unregistering library from pLM+ table 5-30
use of pSOS+ objects 8-6
userid 6-14, 6-67
user_event_id 8-6
ustack 2-203
utime_f 3-90

V

valp 2-268
variable
  condition
    creating 2-31
    deleting 2-33
    obtaining identifier of 2-35
    querying 2-37
    signalling 2-27, 2-39
    waiting on 2-41
  errno
    obtaining address of 2-60
  task
    deleting 2-211
variables
  accessing rpc_createerr 7-8
  accessing svc_fdset 7-7
  socket 6-31
verifying program operation 4-17
verifying volume control structures 3-91
verify_vol 3-91
version 5-8, 5-10, 5-13
vfprintf 4-253
VF_DBDA 3-104
VF_EXTFD 3-103
VF_EXTIN 3-103
VF_INDA 3-104
VF_INFD 3-103
VF_INIX 3-104
VF_IXDA 3-104
VF_IXFD 3-103
VF_LLBFD 3-103
VF_LLBIN 3-103
VF_LLBIX 3-104
VF_TBCFD 3-103
volume
  getting statistics for 3-80
  initializing 3-37
  mounting 3-49
  MS-DOS
    initializing 3-61
    mounting 3-65
  obtaining statistics about 3-31

performing control operations on 3-20
pHILE+
writing directly to 3-109
read directly from 3-73
synchronizing 3-84
unmounting 3-88
verifying control structure 3-91
volume_label 3-38
volume_size 3-38
vprintf 4-255
vqid 2-157
vsprintf 4-257

W

waiting on a condition variable 2-41
wchar 4-261
wcstombs 4-259
wctomb 4-261
whence 4-87
width 6-57
writeset 6-57
write_f 3-107
write_vol 3-109
writing
to a file 3-107
to a stream 4-93
to an I/O device 2-53
writing character
to a stream 4-71, 4-169
to stdout 4-171
writing directly to a pHILE+ formatted volume 3-109
writing formatted output to
a buffer 4-198, 4-257
a stream 4-253
stdout 4-255
writing string to a stream 4-73, 4-172
W_OK 3-7

X

X_OK 3-7

Symbols

_IO_FBF 4-194
_IO_LBF 4-194
_IO_NBF 4-194